BEA WebLogic Server™

Programming WebLogic Web Services

Version 8.1
Revised: June 25, 2004

Copyright

Copyright © 2004 BEA Systems, Inc. All Rights Reserved.

Restricted Rights Legend

This software and documentation is subject to and made available only pursuant to the terms of the BEA Systems License Agreement and may be used or copied only in accordance with the terms of that agreement. It is against the law to copy the software except as specifically allowed in the agreement. This document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine readable form without prior consent, in writing, from BEA Systems, Inc.

Use, duplication or disclosure by the U.S. Government is subject to restrictions set forth in the BEA Systems License Agreement and in subparagraph (c)(1) of the Commercial Computer Software-Restricted Rights Clause at FAR 52.227-19; subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013, subparagraph (d) of the Commercial Computer Software--Licensing clause at NASA FAR supplement 16-52.227-86; or their equivalent.

Information in this document is subject to change without notice and does not represent a commitment on the part of BEA Systems. THE SOFTWARE AND DOCUMENTATION ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND INCLUDING WITHOUT LIMITATION, ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. FURTHER, BEA Systems DOES NOT WARRANT, GUARANTEE, OR MAKE ANY REPRESENTATIONS REGARDING THE USE, OR THE RESULTS OF THE USE, OF THE SOFTWARE OR WRITTEN MATERIAL IN TERMS OF CORRECTNESS, ACCURACY, RELIABILITY, OR OTHERWISE.

Trademarks or Service Marks

BEA, Jolt, Tuxedo, and WebLogic are registered trademarks of BEA Systems, Inc. BEA Builder, BEA Campaign Manager for WebLogic, BEA eLink, BEA Liquid Data for WebLogic, BEA Manager, BEA WebLogic Commerce Server, BEA WebLogic Enterprise, BEA WebLogic Enterprise Platform, BEA WebLogic Express, BEA WebLogic Integration, BEA WebLogic Personalization Server, BEA WebLogic Platform, BEA WebLogic Portal, BEA WebLogic Server, BEA WebLogic Workshop and How Business Becomes E-Business are trademarks of BEA Systems, Inc.

All other trademarks are the property of their respective companies.

Contents

About This Document

Audience xxi
e-docs Web Site xxi
How to Print the Document xxi
Contact Us! xxii
Documentation Conventions xxii

1. Introduction to WebLogic Web Services

Overview of Web Services 1-1
Why Use Web Services? 1-2
Web Service Standards 1-3
BEA Implementation of Web Service Specifications 1-4
SOAP 1-4
WSDL 1.1 1-5
JAX-RPC 1.0 1-6
Web Services Security (WS-Security) 1-7
UDDI 2.0 1-8
Additional Specifications Supported by WebLogic Web Services 1-8
WebLogic Web Service Features 1-8
Unsupported Features 1-11
Examples of Creating and Invoking a Web Service 1-12
Creating WebLogic Web Services: Main Steps 1-13

Roadmap to Common Tasks for Creating Web Services 1-14
Editing XML Files 1-17

2. Architectural Overview

Overview of WebLogic Web Services Architecture 2-1
Back-end Component Operation 2-2
Back-end Component and SOAP Message Handler Chain Operation 2-3
SOAP Message Handler Chain Operation 2-4

3. Creating a WebLogic Web Service: A Simple Example

Overview of the Web Service Example 3-1
Building and Running the Trader WebLogic Web Service Example 3-2
Anatomy of the Example 3-4
The EJB Java Interfaces and Implementation Class 3-4
Remote Interface (Trader.java) 3-4
Session Bean Implementation Class (TraderBean.java) 3-5
Home Interface (TraderHome.java) 3-8
The Non-Built-In Data Type TraderResult 3-8
The EJB Deployment Descriptors 3-9
ejb-jar.xml 3-9
weblogic-ejb-jar.xml 3-10
The servicegen Ant Task That Assembles the Web Service 3-11
The Client Application to Invoke The Web Service 3-11

4. Designing WebLogic Web Services

Choosing the Back-end Components of Your Web Service 4-1
EJB Back-end Component 4-2
Java Class Back-end Component 4-2
Choosing Between Synchronous or Asynchronous Operations 4-2

Choosing RPC-Oriented or Document-Oriented Web Services 4-3
Using Built-In and Non-Built-In Data Types 4-4
Using SOAP Message Handlers to Intercept the SOAP Message. 4-5
Mimicking a Conversational (Stateful) WebLogic Web Service 4-5

5. Implementing WebLogic Web Services

Overview of Implementing a WebLogic Web Service 5-1
Examples of Implementing WebLogic Web Services. 5-2
Implementing a WebLogic Web Service: Main Steps 5-2
Writing the Java Code for the Components. 5-3
Implementing a Web Service By Writing a Stateless Session EJB 5-4
Implementing a Web Service By Writing a Java Class 5-4
Implementing Non-Built-In Data Types. 5-5
Implementing a Document-Oriented Web Service. 5-6
Generating a Partial Implementation From a WSDL File 5-6
Running the wsdl2Service Ant Task 5-7
Sample build.xml Files for the wsdl2Service Ant Task 5-8
Using SOAP Attachments 5-9
java.lang.String 5-10
javax.activation.DataHandler 5-10
Implementing Multiple Return Values 5-10
Using Out and In-Out Parameters 5-11
Using Holder Classes to Implement Multiple Return Values 5-12
Throwing SOAP Fault Exceptions 5-14
Supported Built-In Data Types 5-15
XML Schema-to-Java Mapping for Built-In Data Types. 5-16
Java-to-XML Mapping for Built-In Data Types 5-18

6. Assembling WebLogic Web Services Using Ant Tasks

Overview of Assembling WebLogic Web Services Using Ant Tasks 6-1

Examples of Assembling WebLogic Web Services 6-2

Assembling WebLogic Web Services Using the servicegen Ant Task 6-3

What the servicegen Ant Task Does 6-3

Assembling WebLogic Web Services Automatically: Main Steps 6-3

Creating the Build File That Specifies the servicegen Ant Task 6-4

Assembling WebLogic Web Services Using Individual Ant Tasks 6-5

Assembling a Web Service Starting with Java. 6-6

Assembling a Web Service Starting with an XML Schema. 6-6

Running the source2wsdd Ant Task. 6-8

Running the autotype Ant Task 6-9

Running the clientgen Ant Task. 6-12

Running the wspackage Ant task 6-14

The Web Service EAR File Package. 6-16

Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks 6-16

Supported XML Non-Built-In Data Types 6-17

Supported Java Non-Built-In Data Types 6-19

Data Type Non-Compliance with JAX-RPC 6-20

Non-Roundtripping of Generated Data Type Components. 6-20

Deploying and Testing WebLogic Web Services 6-21

WebLogic Web Services Home Page and WSDL URLs 6-21

Denying Access to the WSDL and Home Page of a WebLogic Web Service 6-23

7. Invoking Web Services from Client Applications and WebLogic Server

Overview of Invoking Web Services 7-2

JAX-RPC API 1.0 7-2

The Runtime Client JAR Files 7-3
Examples of Clients That Invoke Web Services 7-4
Creating Java Client Applications to Invoke Web Services: Main Steps 7-4
Generating the Client JAR File by Running the clientgen Ant Task 7-5
Getting Information About a Web Service 7-7
Writing the Java Client Application to Invoke a Web Service 7-8
Writing a Simple Client Application 7-8
Writing a Client That Uses Out or In-Out Parameters 7-10
Writing an Asynchronous Client Application 7-11
Description of the Generated Asynchronous Web Service Client Stub 7-12
Writing the Asynchronous Client Java Code 7-13
Using Web Services System Properties 7-14
Invoking Web Services from WebLogic Server 7-22
Creating and Using Portable Stubs 7-22
Using the VersionMaker Utility to Update Client JAR Files 7-23
Using a Proxy Server with the WebLogic Web Services Client 7-24
Writing Advanced Java Client Applications 7-25
Writing a Dynamic Client That Uses WSDL 7-25
Writing a Dynamic Client That Does Not Use WSDL 7-27
Writing a Dynamic Client That Uses Non-Built-In Data Types 7-29
Writing a J2ME Client 7-31
Writing a J2ME Client That Uses SSL 7-32

8. Using the WebLogic Web Services APIs

Overview of the WebLogic Web Service APIs 8-1
Registering Data Type Mapping Information in a Dynamic Client 8-2
Accessing HttpSession Information from a Web Service Component 8-5
Introspecting the WSDL from a Client Application 8-6

9. Using JMS Transport to Invoke a WebLogic Web Service

Overview of Using JMS Transport 9-1
Specifying JMS Transport for a WebLogic Web Service: Main Steps. 9-2
Updating the web-services.xml File to Specify JMS Transport 9-3
Invoking a Web Service Using JMS Transport. 9-3

10. Using Reliable SOAP Messaging

Overview of Reliable SOAP Messaging. 10-1
Reliable SOAP Messaging Architecture 10-2
Receiver Transactional Context 10-4
Guidelines For Programming the EJB That Implements a Reliable Web Service Operation 10-5
Guidelines for Programming the Java Class That Implements a Reliable Web Service Operation 10-6
Configuring the Transaction 10-6
Limitations of Reliable SOAP Messaging 10-6
Using Reliable SOAP Messaging: Main Steps 10-6
Configuring the Sender WebLogic Server 10-8
Configuring the Receiver WebLogic Server 10-10
Writing the Java Code to Invoke an Operation Reliably 10-11
Handling Sender Server Failures 10-14
Updating the web-services.xml File Manually for Reliable SOAP Messaging. . . 10-15

11. Using Non-Built-In Data Types

Overview of Using Non-Built-In Data Types 11-1
Creating Non-Built-In Data Types Manually: Main Steps 11-2
Writing the XML Schema Data Type Representation 11-3
Writing the Java Data Type Representation 11-4

Writing the Serialization Class 11-5
Creating the Data Type Mapping File 11-10
Updating the web-services.xml File With XML Schema Information 11-11

12. Creating SOAP Message Handlers to Intercept the SOAP Message

Overview of SOAP Message Handlers and Handler Chains 12-1
Creating SOAP Message Handlers: Main Steps 12-2
Designing the SOAP Message Handlers and Handler Chains 12-4
Implementing the Handler Interface 12-6
Implementing the Handler.init() Method 12-8
Implementing the Handler.destroy() Method 12-8
Implementing the Handler.getHeaders() Method 12-9
Implementing the Handler.handleRequest() Method 12-9
Implementing the Handler.handleResponse() Method 12-10
Implementing the Handler.handleFault() Method 12-11
Directly Manipulating the SOAP Request and Response Message Using SAAJ . 12-12
The SOAPPart Object 12-13
The AttachmentPart Object 12-13
Manipulating Image Attachments in a SOAP Message Handler 12-15
Extending the GenericHandler Abstract Class 12-17
Updating the web-services.xml File with SOAP Message Handler Information 12-19
Using SOAP Message Handlers and Handler Chains in a Client Application 12-21
Accessing the MessageContext of a Handler From the Backend Component 12-23

13. Configuring Security

Overview of Web Services Security 13-1
What Type of Security Should You Configure? 13-2

Configuring Message-Level Security (Digital Signatures and Encryption) 13-3
Main Use Cases 13-3
Unimplemented Features of the Web Services Security Core Specification 13-4
Terminology 13-5
Architectural Overview of Message-Level Security 13-5
Configuring Message-Level Security: Main Steps 13-9
Configuring The Identity Asserter Provider for the myrealm Security Realm 13-12
Updating the servicegen build.xml File 13-12
Updating Security Information in the web-services.xml File 13-14
Encrypting Passwords in the web-services.xml File 13-21
Updating a Java Client to Invoke a Data-Secured Web Service 13-23
Configuring Transport-Level Security (SSL): Main Steps 13-31
Configuring SSL for a Client Application 13-32
Using the WebLogic Server-Provided SSL Implementation 13-33
Configuring the WebLogic SSL Implementation Programatically 13-34
Using SSL Socket Sharing When Using the WebLogic SSL Implementation 13-36
Using a Third-Party SSL Implementation 13-38
Extending the SSLAdapterFactory Class 13-39
Configuring Two-Way SSL For a Client Application 13-40
Using a Proxy Server 13-41
Configuring Access Control Security: Main Steps 13-41
Controlling Access to WebLogic Web Services 13-41
Securing the Entire Web Service and Its Operations 13-42
Securing the Web Service URL 13-43
Securing the Stateless Session EJB and Its Methods 13-43
Securing the WSDL and Home Page of the Web Service 13-43
Specifying the HTTPS Protocol 13-44
Coding a Client Application to Authenticate Itself to a Web Service 13-45

Testing a Secure WebLogic Web Service From Its Home Page 13-46

14.Internationalization

Overview of Internationalization 14-1
Internationalizing a WebLogic Web Service 14-2
Specifying the Character Set for a WebLogic Web Service 14-2
Updating the web-services.xml File 14-2
Setting a WebLogic Server System Property 14-3
Order of Precedence of Character Set Configuration Used By WebLogic Server . . 14-3
Invoking a Web Service Using a Specific Character Set 14-4
Setting the Character Set When Invoking a Web Service 14-4
Character Set Settings in HTTP Request Headers Honored by WebLogic Web Services 14-5

15.Using SOAP 1.2

Overview of Using SOAP 1.2 15-1
Specifying SOAP 1.2 for a WebLogic Web Service: Main Steps 15-2
Updating the web-services.xml File Manually 15-3
Invoking a Web Service Using SOAP 1.2 15-3

16.Creating JMS-Implemented WebLogic Web Services

Overview of JMS-Implemented WebLogic Web Services 16-1
Designing JMS-Implemented WebLogic Web Services 16-2
Retrieving and Processing Messages 16-2
Example of Using JMS Components 16-3
Creating JMS-Implemented WebLogic Web Services 16-3
Configuring JMS Components for Message-Style Web Services 16-4
Assembling JMS-Implemented WebLogic Web Services Using servicegen 16-5
Assembling JMS-Implemented WebLogic Web Services Manually 16-7

Packaging the JMS Message Consumers and Producers 16-8
Updating the web-services.xml File With Component Information 16-8
Sample web-services.xml File for JMS Component Web Service 16-9
Deploying JMS-Implemented WebLogic Web Services 16-10
Invoking JMS-Implemented WebLogic Web Services. 16-10
Invoking an Asynchronous Web Service Operation to Send Data 16-11
Invoking a Synchronous Web Service Operation to Send Data 16-13

17.Administering WebLogic Web Services

Overview of Administering WebLogic Web Services 17-1
Using the Administration Console to Administer Web Services 17-2

18.Publishing and Finding Web Services Using UDDI

Overview of UDDI 18-1
UDDI and Web Services 18-2
UDDI and Business Registry 18-2
UDDI Data Structure 18-3
WebLogic Server UDDI Features 18-4
UDDI 2.0 Server 18-5
Configuring the UDDI 2.0 Server 18-5
Description of Properties in the uddi.properties File 18-6
UDDI Directory Explorer 18-15
UDDI Client API. 18-16
Pluggable tModel 18-16
XML Elements and Permissible Values. 18-17
XML Schema for Pluggable tModels 18-18
Sample XML for a Pluggable tModel 18-20

19.Interoperability

Overview of Interoperability 19-1
Avoid Using Vendor-Specific Extensions 19-2
Stay Current With the Latest Interoperability Tests 19-2
Understand the Data Models of Your Applications 19-3
Understand the Interoperability of Various Data Types 19-3
Results of SOAPBuilders Interoperability Lab Round 3 Tests 19-5
Interoperating With .NET 19-5

20.Troubleshooting

Using the Web Service Home Page to Test Your Web Service 20-2
    URL Used to Invoke the Web Service Home Page 20-2
    Testing the Web Service 20-4
Viewing SOAP Messages 20-4
    Setting Verbose Mode with Ant 20-4
    Setting Verbose Mode Programatically 20-5
Posting the HTTP SOAP Message 20-5
    Composing the SOAP Request 20-7
Debugging Problems with WSDL 20-8
Verifying a WSDL File 20-9
Verifying an XML Schema 20-10
Debugging Data Type Generation (Autotyping) Problems 20-10
    Common XML Schema Problems 20-10
    Common Java Problems 20-11
Debugging Performance Problems 20-11
Performance Hints 20-12
Re-Resolving IP Addresses in the Event of a Failure 20-12
BindingException When Running clientgen or autotype Ant Task 20-13

Client Error When Using the WebLogic Web Service Client to Connect to a Third-Party SSL Server 20-13
Client Error When Invoking Operation That Returns an Abstract Type 20-14
Including Nillable, Optional, and Empty XML Elements in SOAP Messages. 20-15
SSLKeyException When Trying to Invoke a Web Service Using HTTPS 20-17
Autotype Ant Task Not Generating Serialization Classes for All Specified Java Types20-17
Client Gets HTTP 401 Error When Invoking a Non-Secure Web Service. 20-18
Asynchronous Web Service Client Using JMS Transport Not Receiving Response Messages From WebLogic Server 20-19
Running autotype Ant Task on a Large WSDL File Returns java.lang.OutOfMemoryError 20-20
Error When Trying to Log Onto the UDDI Explorer 20-20

21.Upgrading WebLogic Web Services

Overview of Upgrading WebLogic Web Services 21-1
Upgrading a 7.0 WebLogic Web Service to 8.1 21-1
Upgrading a 6.1 WebLogic Web Service to 8.1 21-2
Converting a 6.1 build.xml file to 8.1 21-3
Updating the URL Used to Access the Web Service 21-5

22.Using WebLogic Workshop With WebLogic Web Services

Overview of WebLogic Workshop and WebLogic Web Services 22-1
WebLogic Workshop and WebLogic Web Services. 22-1
EJBGen 22-2
Using Meta-Data Tags When Creating EJBs and Web Services 22-3
Using WebLogic Workshop To Create a WebLogic Web Service: A Simple Example 22-4
Using WebLogic Workshop To Create a WebLogic Web Service: A More Complex Example 22-7
Description of the Example 22-7

Assumptions 22-7
The Example 22-8
Sample build.xml File 22-13
Source Code for Supporting Java Objects 22-16
Item.java 22-17
PurchaseOrder.java 22-18
PurchasingManagerBean.java 22-20
PurchaseOrderFactory.java 22-20

A. WebLogic Web Service Deployment Descriptor Elements

Overview of web-services.xml A-1
Graphical Representation A-1
Element Reference A-4
clock-precision A-5
clocks-synchronized A-5
components A-5
ejb-link A-6
encryptionKey A-6
enforce-precision A-7
fault A-7
generate-signature-timestamp A-7
handler A-8
handler-chain A-8
handler-chains A-8
inbound-expiry A-9
init-param A-9
init-params A-9
java-class A-10

jms-receive-queue A-10
jms-send-destination A-11
jndi-name A-11
name A-11
operation A-12
operations A-15
outbound-expiry A-15
param A-16
params A-18
password A-19
reliable-delivery A-19
require-signature-timestamp A-20
return-param A-21
security A-23
signatureKey A-23
spec:BinarySecurityTokenSpec A-24
spec:ElementIdentifier A-24
spec:EncryptionSpec A-25
spec:SecuritySpec A-27
spec:SignatureSpec A-28
spec:UsernameTokenSpec A-29
stateless-ejb A-30
timestamp A-30
type-mapping A-31
type-mapping-entry A-31
types A-33
user A-33
web-service A-34

web-services A-38

B. Web Service Ant Tasks and Command-Line Utilities

Overview of WebLogic Web Services Ant Tasks and Command-Line Utilities B-1
List of Web Services Ant Tasks and Command-Line Utilities. B-2
Using the Web Services Ant Tasks. B-3
Differences in Operating System Case Sensitivity When Manipulating WSDL and XML Schema Files B-4
Setting the Classpath for the WebLogic Ant Tasks B-5
Using the Web Services Command-Line Utilities B-6
autotype B-7
clientgen. B-14
servicegen B-25
servicegen B-27
service B-29
client B-36
handlerChain B-38
reliability B-39
security B-40
source2wsdd B-41
wsdl2Service B-46
wsdlgen B-50
wspackage B-52

C. source2wsdd Tag Reference

Overview of Using source2wsdd Tags C-1
@wlws:webservice C-2
@wlws:operation. C-5
@wlws:part partname C-7

@wlws:exclude C-11

D. Customizing WebLogic Web Services

Publishing a Static WSDL File D-1
Creating a Custom WebLogic Web Service Home Page D-2

E. Assembling a WebLogic Web Service Manually

Overview of Assembling a WebLogic Web Service Manually E-1
Assembling a WebLogic Web Service Manually: Main Steps E-2
Understanding the web-services.xml File E-2
Creating the web-services.xml File Manually: Main Steps E-3
Creating the <components> Element E-5
Creating <operation> Elements E-6
Specifying the Type of Operation E-6
Specifying the Parameters and Return Value of the Operation E-8
Examining Different Types of web-services.xml Files E-9
EJB Component Web Service with Built-In Data Types E-9
EJB Component Web Service with Non-Built-In Data Types E-10
EJB Component and SOAP Message Handler Chain Web Service E-13
SOAP Message Handler Chain Web Service E-14

About This Document

This document describes BEA WebLogic® Web Services and describes how to develop them and invoke them from a client application.

The document is organized as follows:

- Chapter 1, "Introduction to WebLogic Web Services," provides conceptual information about Web Services in general and the features of WebLogic Web Services.
- Chapter 2, "Architectural Overview," provides an architectural overview of WebLogic Web Services.
- Chapter 3, "Creating a WebLogic Web Service: A Simple Example," describes the end-to-end process of creating a simple WebLogic Web Service based on a stateless session EJB.
- Chapter 4, "Designing WebLogic Web Services," describes the design issues you should consider before developing a WebLogic Web Service.
- Chapter 5, "Implementing WebLogic Web Services," describes how to create the back-end components that implement a Web Service.
- Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks," describes how to use the WebLogic Web Services Ant tasks to automatically generate the final parts of a Web Service (such as the serialization information for non-built-in data types and client JAR file), package them all together into a deployable EAR file, and deploy the EAR file on WebLogic Server.

- Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server," describes how to write a client application that invokes WebLogic Web Services.
- Chapter 8, "Using the WebLogic Web Services APIs," describes how to use the WebLogic Web Services APIs in your client applications.
- Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service," describes how to configure your Web Service so that client applications can use JMS, rather than the default HTTP/S, as the transport when invoking a Web Service.
- Chapter 10, "Using Reliable SOAP Messaging," describes how you can asynchronously and reliably invoke a Web Service running on another WebLogic Server instance.
- Chapter 11, "Using Non-Built-In Data Types," describes how to create the serializers and deserializers that convert user-defined data types between their XML and Java representations.
- Chapter 12, "Creating SOAP Message Handlers to Intercept the SOAP Message," describes how to create handlers that intercept a SOAP message for further processing.
- Chapter 13, "Configuring Security," describes how to configure security for WebLogic Web Services.
- Chapter 14, "Internationalization," describes how to specify the character set for a WebLogic Web Service and in a client application that invokes a Web Service.
- Chapter 15, "Using SOAP 1.2," describes how you can use SOAP 1.2, rather than the default SOAP 1.1. as the message format when invoking a WebLogic Web Service.
- Chapter 16, "Creating JMS-Implemented WebLogic Web Services," describes how to create a WebLogic Web Service that is implemented with a JMS message consumer or producer.
- Chapter 17, "Administering WebLogic Web Services," describes how to use the Administration Console to administer WebLogic Web Services.
- Chapter 18, "Publishing and Finding Web Services Using UDDI," describes how to use the UDDI features included in WebLogic Server.
- Chapter 19, "Interoperability," describes what it means for Web Services to interoperate with each other and provides tips for creating highly interoperable Web Services.
- Chapter 20, "Troubleshooting," describes how to troubleshoot problems with programming or invoking Web Services.

- Chapter 21, "Upgrading WebLogic Web Services," describes how to upgrade Web Services created in Version 6.1 or 7.0 of WebLogic Server to Version 8.1.
- Chapter 22, "Using WebLogic Workshop With WebLogic Web Services," describes how to use WebLogic Workshop with WebLogic Web Services.
- Appendix A, "WebLogic Web Service Deployment Descriptor Elements," describes the elements in the Web Services deployment descriptor file, `web-services.xml`.
- Appendix B, "Web Service Ant Tasks and Command-Line Utilities," describes the Ant tasks, along with their equivalent command-line utilities, used to assemble WebLogic Web Services.
- Appendix C, "source2wsdd Tag Reference," describes the WebLogic Web Service metadata tags you can include in the Java code that implements a Web Service. The `source2wsdd` Ant task uses these tags when generating the Web Service deployment descriptor file (`web-services.xml`).
- Appendix D, "Customizing WebLogic Web Services," describes how to customize WebLogic Web Services by updating the Web Application's `web.xml` deployment descriptor file.
- Appendix E, "Assembling a WebLogic Web Service Manually," describes how assemble a WebLogic Web Service manually without using the WebLogic Web Services Ant tasks.

Audience

This document is written for Java developers who want to create a Web Service that runs on WebLogic Server.

It is assumed that readers know Web technologies, XML, and the Java programming language.

e-docs Web Site

BEA product documentation is available on the BEA corporate Web site. From the BEA Home page, click on Product Documentation.

How to Print the Document

You can print a copy of this document from a Web browser, one main topic at a time, by using the File—Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click Download Documentation, and select the document you want to print.

Adobe Acrobat Reader is available at no charge from the Adobe Web site at http://www.adobe.com.

Contact Us!

Your feedback on BEA documentation is important to us. Send us e-mail at docsupport@bea.com if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the documentation.

In your e-mail message, please indicate the software name and version you are using, as well as the title and document date of your documentation. If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSupport at http://www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes
- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Usage
Ctrl+Tab	Keys you press simultaneously.
italics	Emphasis and book titles.

`monospace text`	Code samples, commands and their options, Java classes, data types, directories, and file names and their extensions. Monospace text also indicates text that you enter from the keyboard.
	Examples:
	`import java.util.Enumeration;`
	`chmod u+w *`
	`config/examples/applications`
	`.java`
	`config.xml`
	`float`

`monospace italic text`	Variables in code. *Example*: `String CustomerName;`
UPPERCASE TEXT	Device names, environment variables, and logical operators. *Example*s: LPT1 BEA_HOME OR
`{ }`	A set of choices in a syntax line.
`[ ]`	Optional items in a syntax line. *Example*: `java utils.MulticastTest -n name -a address [-p portnumber] [-t timeout] [-s send]`
`|`	Separates mutually exclusive choices in a syntax line. *Example*: `java weblogic.deploy [list|deploy|undeploy|update] password {application} {source}`

Convention	Usage
. . .	Indicates one of the following in a command line: • An argument can be repeated several times in the command line. • The statement omits additional optional arguments. • You can enter additional parameters, values, or other information
. . .	Indicates the omission of items from a code example or from a syntax line.

CHAPTER 1

Introduction to WebLogic Web Services

The following sections provide an overview of Web Services, and briefly describe how they are implemented in WebLogic Server:

- "Overview of Web Services" on page 1-1
- "Why Use Web Services?" on page 1-2
- "Web Service Standards" on page 1-3
- "WebLogic Web Service Features" on page 1-8
- "Unsupported Features" on page 1-11
- "Examples of Creating and Invoking a Web Service" on page 1-12
- "Creating WebLogic Web Services: Main Steps" on page 1-13
- "Roadmap to Common Tasks for Creating Web Services" on page 1-14
- "Editing XML Files" on page 1-17

Overview of Web Services

Web Services are a special type of service that can be shared by and used as components of distributed Web-based applications. They commonly interface with existing back-end applications, such as customer relationship management systems, order-processing systems, and so on.

Traditionally, software application architecture tended to fall into two categories: huge monolithic systems running on mainframes or client-server applications running on desktops. Although these architectures work well for the purpose the applications were built to address, they are closed and can not be easily accessed by the diverse users of the Web.

Thus the software industry has evolved toward loosely coupled service-oriented applications that interact dynamically over the Web. The applications break down the larger software system into smaller modular components, or shared services. These services can reside on different computers and can be implemented by vastly different technologies, but they are packaged and transported using standard Web protocols, such as XML and HTTP, thus making them easily accessible by any user on the Web.

This concept of services is not new—RMI, COM, and CORBA are all service-oriented technologies. However, applications based on these technologies require them to be written using that particular technology, often from a particular vendor. This requirement typically hinders widespread acceptance of an application on the Web. To solve this problem, Web Services are defined to share the following properties that make them easily accessible from heterogeneous environments:

- Web Services are accessed over the Web.
- Web Services describe themselves using an XML-based description language.
- Web Services communicate with clients (both end-user applications or other Web Services) through XML messages that are transmitted by standard Internet protocols, such as HTTP.

Why Use Web Services?

Major benefits of Web Services include:

- Interoperability among distributed applications that span diverse hardware and software platforms
- Easy, widespread access to applications through firewalls using Web protocols
- A cross-platform, cross-language data model (XML) that facilitates developing heterogeneous distributed applications

Because you access Web Services using standard Web protocols such as XML and HTTP, the diverse and heterogeneous applications on the Web (which typically already understand XML and HTTP) can automatically access Web Services, and thus communicate with each other.

These different systems can be Microsoft SOAP ToolKit clients, J2EE applications, legacy applications, and so on. They are written in Java, C++, Perl, and other programming languages. Application interoperability is the goal of Web Services and depends upon the service provider's adherence to published industry standards.

Web Service Standards

A Web Service requires the following standard implementations:

- An implementation hosted by a server on the Web.

  WebLogic Web Services are hosted by WebLogic Server; are implemented using standard J2EE components (such as Enterprise Java Beans) and Java classes; and are packaged as standard J2EE Enterprise Applications.

- A standard for transmitting data and Web Service invocation calls between the Web Service and the user of the Web Service.

  WebLogic Web Services use Simple Object Access Protocol (SOAP) 1.1 and 1.2 as the message format and HTTP and JMS as the connection protocol. See "SOAP" on page 1-4.

- A standard for describing the Web Service to clients so they can invoke it.

  WebLogic Web Services use Web Services Description Language (WSDL) 1.1, an XML-based specification, to describe themselves. See "WSDL 1.1" on page 1-5.

- A standard for client applications to invoke a Web Service.

  WebLogic Web Services implement the Java API for XML-based RPC (JAX-RPC) 1.0 as part of a client JAR that client applications can use to invoke WebLogic and non-WebLogic Web Services. See "JAX-RPC 1.0" on page 1-6.

- A standard for digitally signing and encrypting the SOAP request and response messages between a client application and the Web Service it is invoking.

  WebLogic Web Services implement the following OASIS Standard 1.0 Web Services Security specifications, dated April 6 2004:

  – Web Services Security: SOAP Message Security

  – Web Services Security: Username Token Profile

  – Web Services Security: X.509 Token Profile

  For more information, see "Web Services Security (WS-Security)" on page 1-7.

- A standard for client applications to find a registered Web Service and to register a Web Service.

  WebLogic Web Services implement the Universal Description, Discovery, and Integration (UDDI) specification. See "UDDI 2.0" on page 1-8.

BEA Implementation of Web Service Specifications

Many of the specifications that define Web Service standards have been written in an intentionally vague way to allow for broad use of the specification throughout the industry. Because of this vagueness, BEA's implementation of a particular specification might not cover all possible usage scenarios covered by the specification.

BEA considers interoperability of Web Services platforms to be more important than providing support for all possible edge cases of the Web Services specifications. For this reason, BEA fully supports the Basic Profile 1.0 specification from the Web Services Interoperability Organization and considers it to be the baseline for Web Services interoperability. BEA implements all requirements of the Basic Profile 1.0, although this guide does not necessarily document all of these requirements. This guide does, however, document features that are beyond the requirements of the Basic Profile 1.0."

SOAP

SOAP (Simple Object Access Protocol) is a lightweight XML-based protocol used to exchange information in a decentralized, distributed environment. WebLogic Server includes its own implementation of SOAP 1.1, SOAP 1.2, and SOAP With Attachments (SAAJ) specifications. The protocol consists of:

- An envelope that describes the SOAP message. The envelope contains the body of the message, identifies who should process it, and describes how to process it.
- A set of encoding rules for expressing instances of application-specific data types.
- A convention for representing remote procedure calls and responses.

This information is embedded in a Multipurpose Internet Mail Extensions (MIME)-encoded package that can be transmitted over HTTP or other Web protocols. MIME is a specification for formatting non-ASCII messages so that they can be sent over the Internet.

The following example shows a SOAP request for stock trading information embedded inside an HTTP request:

```
POST /StockQuote HTTP/1.1
Host: www.sample.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
         SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <SOAP-ENV:Body>
          <m:GetLastStockQuote xmlns:m="Some-URI">
               <symbol>BEAS</symbol>
          </m:GetLastStockQuote>
     </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

For more information, see SOAP 1.1 at http://www.w3.org/TR/SOAP and SOAP With Attachments API for Java (SAAJ) 1.1 at http://java.sun.com/xml/saaj/index.html.

WSDL 1.1

Web Services Description Language (WSDL) is an XML-based specification that describes a Web Service. A WSDL document describes Web Service operations, input and output parameters, and how a client application connects to the Web Service.

Developers of WebLogic Web Services do not need to create the WSDL files; you generate these files automatically as part of the WebLogic Web Services development process.

The following example, for informational purposes only, shows a WSDL file that describes the stock trading Web Service StockQuoteService that contains the method GetLastStockQuote:

```
<?xml version="1.0"?>
  <definitions name="StockQuote"
               targetNamespace="http://sample.com/stockquote.wsdl"
               xmlns:tns="http://sample.com/stockquote.wsdl"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
               xmlns:xsd1="http://sample.com/stockquote.xsd"
               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
               xmlns="http://schemas.xmlsoap.org/wsdl/">
      <message name="GetStockPriceInput">
          <part name="symbol" element="xsd:string"/>
      </message>
      <message name="GetStockPriceOutput">
          <part name="result" type="xsd:float"/>
      </message>
      <portType name="StockQuotePortType">
          <operation name="GetLastStockQuote">
             <input message="tns:GetStockPriceInput"/>
```

```
            <output message="tns:GetStockPriceOutput"/>
         </operation>
      </portType>
      <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
         <soap:binding style="rpc"
                       transport="http://schemas.xmlsoap.org/soap/http"/>
         <operation name="GetLastStockQuote">
            <soap:operation soapAction="http://sample.com/GetLastStockQuote"/>
            <input>
             <soap:body use="encoded" namespace="http://sample.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
            <soap:body use="encoded" namespace="http://sample.com/stockquote"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
         </operation>>
      </binding>
      <service name="StockQuoteService">
         <documentation>My first service</documentation>
         <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
            <soap:address location="http://sample.com/stockquote"/>
         </port>
      </service>
  </definitions>
```

For more information, see Web Services Description Language (WSDL) 1.1 at http://www.w3.org/TR/wsdl.

JAX-RPC 1.0

The Java API for XML-based RPC (JAX-RPC) 1.0is a Sun Microsystems specification that defines the Web Services APIs.

WebLogic Server implements all required features of the JAX-RPC Version 1.0 specification. Additionally, WebLogic Server implements optional data type support, as specified in:

- "Supported Built-In Data Types" on page 5-15
- "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16

WebLogic Server does not implement optional features of the JAX-RPC specification, other than what is described in these sections.

The following table briefly describes the core JAX-RPC interfaces and classes.

Table 1-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface. Used for both static and dynamic invocations.
ServiceFactory	Factory class for creating `Service` instances.
Stub	Represents the client proxy for invoking the operations of a Web Service. Typically used for static invocation of a Web Service.
Call	Used to invoke a Web Service dynamically.
JAXRPCException	Exception thrown if an error occurs while invoking a Web Service.

For detailed information on JAX-RPC, see http://java.sun.com/xml/jaxrpc/index.html.

For a tutorial that describes how to use JAX-RPC to invoke Web Services, see http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html.

Web Services Security (WS-Security)

The following description of Web Services Security is taken directly from the OASIS standard 1.0 specification, titled *Web Services Security: SOAP Message Security*, dated April 6, 2004:

> This specification proposes a standard set of SOAP extensions that can be used when building secure Web services to implement integrity and confidentiality. We refer to this set of extensions as the *Web Services Security Language* or *WS-Security.*
>
> WS-Security is flexible and is designed to be used as the basis for the construction of a wide variety of security models including PKI, Kerberos, and SSL. Specifically WS-Security provides support for multiple security tokens, multiple trust domains, multiple signature formats, and multiple encryption technologies.
>
> This specification provides three main mechanisms: security token propagation, message integrity, and message confidentiality. These mechanisms by themselves do not provide a complete security solution. Instead, WS-Security is a building block that can be used in conjunction with other Web service extensions and higher-level application-specific protocols to accommodate a wide variety of security models and encryption technologies.

These mechanisms can be used independently (for example, to pass a security token) or in a tightly integrated manner (for example, signing and encrypting a message and providing a security token hierarchy associated with the keys used for signing and encryption).

For more information, see the OASIS Web Service Security Web page at http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

UDDI 2.0

The Universal Description, Discovery and Integration (UDDI) specification defines a standard for describing a Web Service; registering a Web Service in a well-known registry; and discovering other registered Web Services.

For more information, see http://www.uddi.org.

Additional Specifications Supported by WebLogic Web Services

- XML Schema Part 1: Structures at http://www.w3.org/TR/xmlschema-1/
- XML Schema Part 2: Data Types at http://www.w3.org/TR/xmlschema-2/
- JSSE (part of JDK 1.4) at http://java.sun.com/products/jsse

WebLogic Web Service Features

The WebLogic Web Services subsystem has the following features (new features in Version 8.1 of WebLogic Server are listed first):

- **Digital Signatures and Encryption - New 8.1 Feature**

  New elements in the `web-services.xml` deployment descriptor enable you to configure message-level security for Web Services and Web Service clients. See "Configuring Message-Level Security: Main Steps" on page 13-9.

- **Reliable SOAP Messaging - New 8.1 Feature**

  Reliable SOAP messaging is a framework whereby an application running in one WebLogic Server instance can asynchronously and reliably invoke a Web Service running on another WebLogic Server instance. See Chapter 10, "Using Reliable SOAP Messaging."

- **Asynchronous Client Invocation of WebLogic Web Services - New 8.1 Feature**

The `clientgen` Ant task can now generate stubs for invoking a Web Service operation asynchronously. The stub contains two methods: the first invokes the operation with the required parameters but does not wait for the result to return; later, the second method returns the actual results. You use this asynchronous client when using reliable SOAP messaging. See "Writing an Asynchronous Client Application" on page 7-11.

- **JMS Transport Protocol - New 8.1 Feature**

  You can configure a Web Service to use JMS as the transport protocol (as opposed to HTTP/S, the default protocol) when a client accesses the service. See Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service."

- **Portable Stubs - New 8.1 Feature**

  You can now use portable stubs (versioned client JAR files used to invoke WebLogic Web Services) to avoid class clashes when invoking a Web Service from within WebLogic Server. See "Creating and Using Portable Stubs" on page 7-22.

- **Implementation of the SOAP with Attachments API For Java (SAAJ) 1.1 - New 8.1 Feature**

  SAAJ enables developers to produce and consume messages conforming to the SOAP 1.1 specification and SOAP with Attachments note. This specification is derived from the `javax.xml.soap` package originally defined in the JAXM 1.0 specification.

  See "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 12-12 for information about using SAAJ in a SOAP message handler to view and manipulate a SOAP attachment.

- **SOAP 1.2 Support - New 8.1 Feature**

  WebLogic Server provides support for using SOAP 1.2 as the message format when a client invokes a Web Service operation. See Chapter 15, "Using SOAP 1.2."

- **Standard Specifications**

  See "Web Service Standards" on page 1-3.

- **Support for Exposing Standard J2EE Components**

  WebLogic Web Services support exposing standard J2EE components, such as stateless session EJBs.

- **Ant Tasks and Command Line Utilities**

  Ant tasks facilitate the implementation and packaging of Web Services. See Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

- **UDDI Registry, Directory Explorer, and Client API**

  WebLogic Server includes a UDDI registry, a UDDI Directory Explorer, and an implementation of the UDDI client API. See Chapter 18, "Publishing and Finding Web Services Using UDDI."

- **Support for Both RPC-Oriented and Document-Oriented Operations**

  WebLogic Web Service operations can be either RPC-oriented (SOAP messages contain parameters and return values) or document-oriented (SOAP messages contain documents.) For details, see "Choosing RPC-Oriented or Document-Oriented Web Services" on page 4-3.

- **Support for User-Defined Data Types**

  You can create a WebLogic Web Service that uses non-built-in data types as its parameters and returns values. Non-built-in data types are defined as data types other than the supported built-in data types; examples of built-in data types include `int` and `String`. WebLogic Server Ant tasks can generate the components needed to use non-built-in data types; this feature is referred to as *autotyping*. You can also create these components manually. See Appendix B, "Web Service Ant Tasks and Command-Line Utilities," and Chapter 11, "Using Non-Built-In Data Types."

- **SOAP Message Handlers to Access SOAP Messages**

  A SOAP message handler accesses the SOAP message and its attachment in both the request and response of the Web Service. You can create handlers in both the Web Service itself and the client applications that invoke the Web Service. See Chapter 12, "Creating SOAP Message Handlers to Intercept the SOAP Message."

- **Java Client to Invoke a Web Service**

  Developers can use an automatically generated thin Java client to create Java client applications that invoke WebLogic and non-WebLogic Web Services. See Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server."

  Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

- **The Web Services Home Web Page**

  All deployed WebLogic Web Services automatically have a Home Web Page that includes links to the WSDL of the Web Service, the client JAR file that you can download for invoking the Web Service, and a mechanism for testing the Web Service to ensure that it is working as expected. See "WebLogic Web Services Home Page and WSDL URLs" on page 6-21.

- **Point-to-Point Security**

  WebLogic Server supports connection oriented point-to-point security for WebLogic Web Service operations, as well as authorization and authentication of Web Service operations. See "Configuring Transport-Level Security (SSL): Main Steps" on page 13-31.

- **Interoperability**

  WebLogic Web Services interoperate with major Web Service platforms such as Microsoft .NET.

- **Java 2 Platform Micro Edition (J2ME) Clients**

  The WebLogic Server the `clientgen` Ant task can create a client JAR file that runs on J2ME. See Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server."

Unsupported Features

WebLogic Server does not support the following XML Schema features:

- Complex data type inheritance by restriction
- Union simple data types
- References to named model groups
- Nested content models in a single complex type
- Redefinition of declarations
- Identity constraints (key, keyref, unique)
- Wildcards
- Substitution groups

Note: If you use the `autotype`, `servicegen`, or `clientgen` Ant tasks to generate the serialization components for any non-built-in XML Schema data type that uses one of the preceding constructs (either directly or by containing a type that uses them), the Ant tasks map that data type to `javax.xml.soap.SOAPElement`. This gives you access to the full XML via a DOM-like API.

WebLogic Server does not support the following WSDL features:

- Overloading operations in WSDL, due to a SOAP limitation
- HTTP GET and POST bindings
- Faults with complex types
- RPC literal style
- Document encoded style
- `solicit-response` and `notification` transmission primitives

Examples of Creating and Invoking a Web Service

WebLogic Server includes the following examples of creating and invoking WebLogic Web Services in the *`WL_HOME`*`/samples/server/examples/src/examples/webservices` directory, where *`WL_HOME`* refers to the main WebLogic Platform directory:

- `basic.statelessSession`: Uses a stateless session EJB back-end component with built-in data types as its parameters and return value.
- `basic.javaclass`: Uses a Java class back-end component with built-in data types as its parameters and return value.
- `complex.statelessSession`: Uses a stateless session EJB back-end component with non-built-in data types as its parameters and return value.
- `handler.log`: Uses both a handler chain and a stateless session EJB.
- `handler.nocomponent`: Uses only a handler chain with *no* back-end component.
- `client.static_no_out`: Shows how to create a static client application that invokes a non-WebLogic Web Service.
- `client.dynamic_wsdl`: Shows how to create a dynamic client application that uses WSDL to invoke a non-WebLogic Web Service.
- `client.dynamic_no_wsdl`: Shows how to create a dynamic client application that does not use WSDL to invoke a non-WebLogic Web Service.

For detailed instructions on how to build and run the examples, open the following Web page in your browser:

`WL_HOME``/samples/server/examples/src/examples/webservices/package-summary.html`

Creating WebLogic Web Services: Main Steps

The following procedure describes the high-level steps to create a WebLogic Web Service. Most steps are described in detail in later chapters. Chapter 3, "Creating a WebLogic Web Service: A Simple Example," briefly describes an example of creating a Web Service.

1. Design the WebLogic Web Service.

   Decide on an RPC- or document-oriented Web Service; a synchronous or asynchronous Web Service; the type of back-end components that implement the service; whether your service uses only built-in data types or custom data types; whether you need to intercept the incoming or outgoing SOAP message; and so on.

   See Chapter 4, "Designing WebLogic Web Services."

2. Implement the basic WebLogic Web Service.

   Write and compile the Java code of the back-end components that make up the Web Service; optionally create SOAP message handlers that intercept the SOAP messages; optionally create your own serialization class to convert data between XML and Java; and so on.

   See Chapter 5, "Implementing WebLogic Web Services."

3. Assemble and package the WebLogic Web Service.

   Gather all the implementation class files into an appropriate directory structure; create the Web Service deployment descriptor file; create the supporting parts of the service (such as client JAR file); and package everything into a deployable unit (either an EAR file or in exploded directory format).

   If your Web Service is fairly simple, use the `servicegen` Ant task, which performs all the assembly steps for you. If your Web Service is more complicated, use individual Ant tasks.

   See Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

4. Deploy the basic WebLogic Web Service for testing purposes.

   Make the service available to remote clients. Because WebLogic Web Services are packaged as standard J2EE Enterprise Applications, deploying a Web Service is the same as deploying an Enterprise Application.

   See *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/deployment/index.html.

5. Create a client that accesses the Web Service to test that your Web Service is working as you expect. You can also use the Web Service Home Page to test your Web Service.

See Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server."

6. Configure additional WebLogic Web Service features, such as security, reliable SOAP messaging, JMS transport, internationalization, and so on. See:
   - Chapter 13, "Configuring Security"
   - Chapter 10, "Using Reliable SOAP Messaging"
   - Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service"
   - Chapter 14, "Internationalization"
7. Test the WebLogic Web Service after you add features.

   See "Deploying and Testing WebLogic Web Services" on page 6-21.
8. Deploy the WebLogic Web Service for production.

   See *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/deployment/index.html.
9. Optionally publish your Web Service in a UDDI registry. See Chapter 18, "Publishing and Finding Web Services Using UDDI."

Roadmap to Common Tasks for Creating Web Services

The following table provides a roadmap of common tasks for creating, deploying, and invoking WebLogic Web Services

.

Table 1-2 Web Services Tasks

Major Task	Subtasks and Additional Information
Create (implement) the Web Service back-end components.	"Overview of Implementing a WebLogic Web Service" on page 5-1
	"Writing the Java Code for the Components" on page 5-3
	"Generating a Partial Implementation From a WSDL File" on page 5-6
	"Creating SOAP Message Handlers to Intercept the SOAP Message" on page 12-1
	"Using SOAP Attachments" on page 5-9
	"Using Built-In and Non-Built-In Data Types" on page 4-4
	"Implementing Non-Built-In Data Types" on page 5-5
	"Supported Built-In Data Types" on page 5-15
	"Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16
	"Throwing SOAP Fault Exceptions" on page 5-14
	"Using the WebLogic Web Services APIs" on page 8-1
Assemble the Web Service into a deployable unit.	"Overview of Assembling WebLogic Web Services Using Ant Tasks" on page 6-1
	"Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3
	"Assembling WebLogic Web Services Using Individual Ant Tasks" on page 6-5
	"Assembling a Web Service Starting with Java" on page 6-6
	"Assembling a Web Service Starting with an XML Schema" on page 6-6

Table 1-2 Web Services Tasks

Major Task	Subtasks and Additional Information
Deploy and test the Web Service.	"Deploying and Testing WebLogic Web Services" on page 6-21
	"WebLogic Web Services Home Page and WSDL URLs" on page 6-21
Invoke the Web Service.	"Overview of Invoking Web Services" on page 7-2
	"Creating Java Client Applications to Invoke Web Services: Main Steps" on page 7-4
	"Writing an Asynchronous Client Application" on page 7-11
	"Invoking Web Services from WebLogic Server" on page 7-22
Secure the Web Service.	"Overview of Web Services Security" on page 13-1
	"Configuring Message-Level Security: Main Steps" on page 13-9
	"Configuring Transport-Level Security (SSL): Main Steps" on page 13-31
	"Configuring Access Control Security: Main Steps" on page 13-41
Add advanced features.	"Using Reliable SOAP Messaging" on page 10-1
	"Internationalization" on page 14-1
Upgrade a 6.1 or 7.0 WebLogic Web Service.	"Upgrading a 7.0 WebLogic Web Service to 8.1" on page 21-1
	"Upgrading a 6.1 WebLogic Web Service to 8.1" on page 21-2

Table 1-2 Web Services Tasks

Major Task	Subtasks and Additional Information
Troubleshoot problems.	"Using the Web Service Home Page to Test Your Web Service" on page 20-2
	"Viewing SOAP Messages" on page 20-4
	"Posting the HTTP SOAP Message" on page 20-5
	"Debugging Problems with WSDL" on page 20-8
	"Verifying a WSDL File" on page 20-9
	"Verifying an XML Schema" on page 20-10
	"Debugging Data Type Generation (Autotyping) Problems" on page 20-10

Editing XML Files

When creating or invoking WebLogic Web Services, you might need to edit XML files, such as the `web-services.xml` Web Services deployment descriptor file, the EJB deployment descriptors, the Java Ant build files, and so on. You edit these files with the BEA XML Editor.

Note: This guide describes how to create or update the `web-services.xml` deployment descriptor manually so that programmers get a better understanding of the file and how the elements describe a Web Service. You can also use the BEA XML Editor to update the file.

The BEA XML Editor is a simple, user-friendly Java-based tool for creating and editing XML files. It displays XML file contents both as a hierarchical XML tree structure and as raw XML code. This dual presentation of the document gives you two options for editing the XML document:

- The hierarchical tree view allows structured, constrained editing, with a set of allowable functions at each point in the hierarchical XML tree structure. The allowable functions are syntactically dictated and in accordance with the XML document's DTD or schema, if one is specified.
- The raw XML code view allows free-form editing of the data.

The BEA XML Editor can validate XML code according to a specified DTD or XML schema.

For detailed information about using the BEA XML Editor, see its online help.

You can download the BEA XML Editor from dev2dev Online at http://dev2dev.bea.com/resourcelibrary/utilitiestools/xml.jsp.

CHAPTER 2

Architectural Overview

The following sections provide an overview of WebLogic Web Services architecture and three types of WebLogic Web Service operations:

- "Overview of WebLogic Web Services Architecture" on page 2-1
- "Back-end Component Operation" on page 2-2
- "Back-end Component and SOAP Message Handler Chain Operation" on page 2-3
- "SOAP Message Handler Chain Operation" on page 2-4

Overview of WebLogic Web Services Architecture

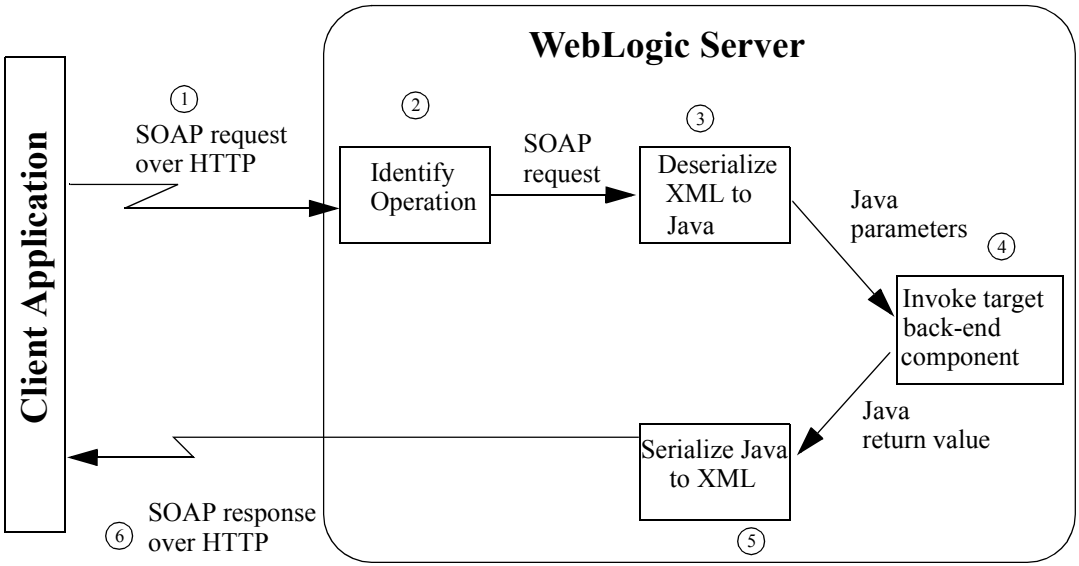
You develop a WebLogic Web Service, by using standard J2EE components, such as stateless session EJBs, and Java classes. Because WebLogic Web Services are based on the J2EE platform, they automatically inherit all the standard J2EE benefits, such as a simple and familiar component-based development model, scalability, support for transactions, life-cycle management, easy access to existing enterprise systems through the use of J2EE APIs (such as JDBC and JTA), and a simple and unified security model.

A single WebLogic Web Service consists of one or more operations; you can implement each operation using different back-end components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

Back-end Component Operation

The following figure describes the architecture of a WebLogic Web Service operation that is implemented with only a back-end component, such as a method of a stateless session EJB.

Figure 2-1 WebLogic Web Service with Back-end Component



Here is what happens when a client application invokes this type of WebLogic Web Service operation:

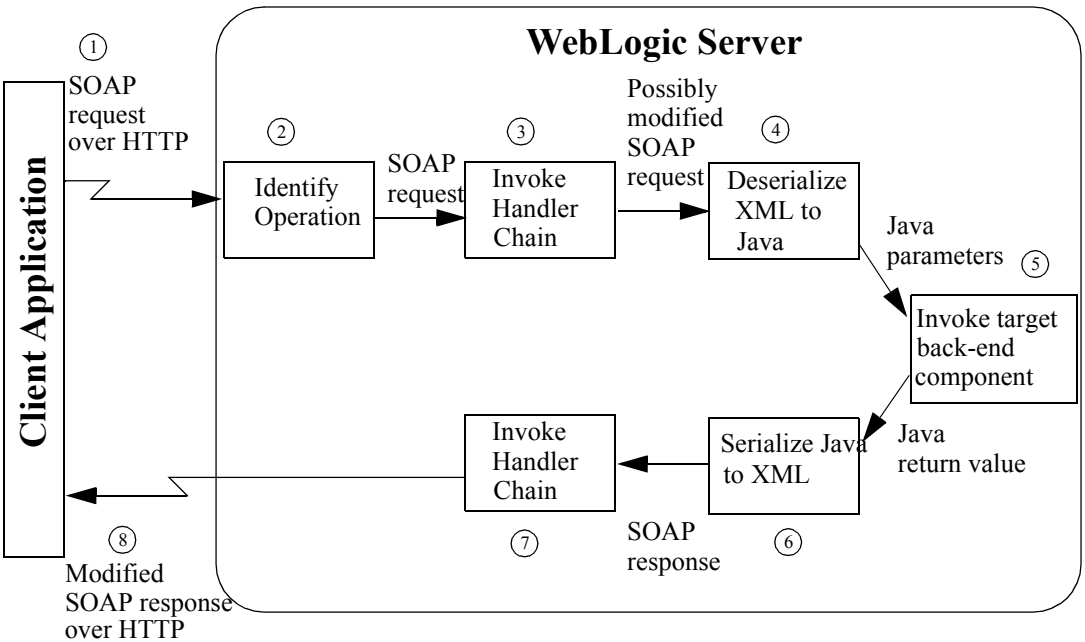
1. The client application sends a SOAP message request to WebLogic Server over HTTP. Based on the URI in the request, WebLogic Server identifies the Web Service being invoked.
2. The Web Service reads the SOAP message request and identifies the operation that it needs to run. The operation corresponds to a method of a stateless session EJB or a Java class, to be invoked in a later step.
3. The Web Service converts the operation's parameters in the SOAP message from their XML representation to their Java representation using the appropriate deserializer class. The deserializer class is either one provided by WebLogic Server for built-in data types or a user-created one for non-built-in data types.
4. The Web Service invokes the appropriate back-end component method, passing it the Java parameters.

5. The Web Service converts the method's return value from Java to XML using the appropriate serializer class, and packages it into a SOAP message response.
6. The Web Service sends the SOAP message response back to the client application that invoked the Web Service.

Back-end Component and SOAP Message Handler Chain Operation

The following figure describes a WebLogic Web Service operation that is implemented with both a SOAP message handler chain and a back-end component.

Figure 2-2 WebLogic Web Service Operation with Back-end Component and SOAP Message Handler Chain



Here is what happens when a client application invokes this type of WebLogic Web Service operation:

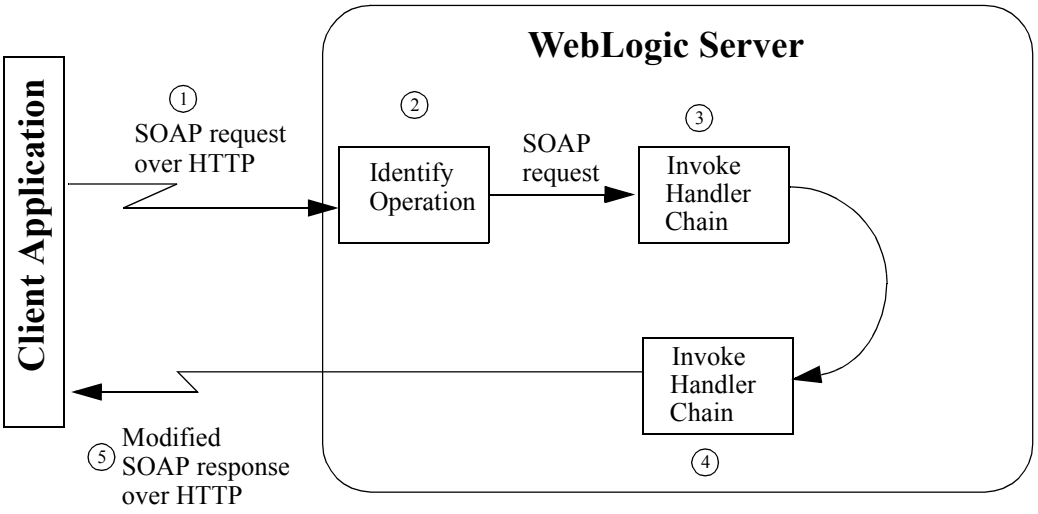
1. The client application sends a SOAP message request to WebLogic Server over HTTP. Based on the URI in the request, WebLogic Server identifies the Web Service being invoked.

2. The Web Service reads the SOAP message request and identifies the operation that it needs to run. The operation in this case corresponds to a SOAP message handler chain followed by a method of a stateless session EJB or a Java class, to be invoked in later steps.
3. The Web Service invokes the appropriate handler chain. The handler chain accesses the contents of the SOAP message request, possibly changing it in some way.
4. The Web Service converts the operation's parameters in the [possibly modified] SOAP message from their XML representation to their Java representation using the appropriate deserializer class. The deserializer class is either one provided by WebLogic Server for built-in data types or a user-created one for non-built-in data types.
5. The Web Service invokes the appropriate back-end component method, passing it the Java parameters.
6. The Web Service converts the method's return value from Java to XML using the appropriate serializer class, and packages it into a SOAP message response.
7. The Web Service invokes the appropriate SOAP message handler chain. The handler chain accesses the contents of the SOAP message response, possibly changing it in some way.
8. The Web Service sends the [possibly modified] SOAP message response back to the client application that invoked the Web Service.

SOAP Message Handler Chain Operation

The following figure describes the architecture of a WebLogic Web Service operation that is implemented with only a SOAP message handler chain.

Figure 2-3 WebLogic Web Service Operation with SOAP Message Handler Chain Only



Here is what happens when a client application invokes this type of WebLogic Web Service operation:

1. The client application sends a SOAP message request to WebLogic Server over HTTP. Based on the URI in the request, WebLogic Server identifies the Web Service being invoked.
2. The Web Service reads the SOAP message request and identifies the operation that it needs to run. The operation in this case corresponds to an invoke of a SOAP message handler chain, to be invoked in the next step.
3. The Web Service invokes the appropriate handler chain. The handler chain accesses the contents of the SOAP message request, possibly changing it in some way.
4. The Web Service invokes the appropriate handler chain. The handler chain creates the SOAP message response.
5. The Web Service sends the SOAP message response back to the client application that invoked the Web Service.

CHAPTER 3

Creating a WebLogic Web Service: A Simple Example

The following sections describe how to create a simple WebLogic Web Service:

- "Overview of the Web Service Example" on page 3-1
- "Building and Running the Trader WebLogic Web Service Example" on page 3-2
- "Anatomy of the Example" on page 3-4

Overview of the Web Service Example

This example describes the start-to-finish process of implementing, assembling, and deploying the WebLogic Web Service provided as a product example in the directory *WL_HOME*`/samples/server/examples/src/examples/webservices/complex/statelessSession`, where *WL_HOME* refers to the main WebLogic Platform directory, such as `d:\beahome\weblogic81`.

The example shows how to create a WebLogic Web Service based on a stateless session EJB. The example uses the `Trader` EJB, one of the EJB 2.0 examples located in the *WL_HOME*`/samples/server/examples/src/examples/ejb20/basic/statelessSession` directory.

The `Trader` EJB defines two methods, `buy()` and `sell()`, that take as input a `String` stock symbol and an `int` number of shares to buy or sell. Both methods return a non-built-in data type called `TradeResult`.

When the `Trader` EJB is converted into a Web Service, the two methods become public operations defined in the WSDL of the Web Service. The `Client.java` application uses a JAX-RPC style client API to create SOAP messages that invoke the operations.

Building and Running the Trader WebLogic Web Service Example

The procedure below describes how to build and run the *`WL_HOME`*`/samples/server/examples/src/examples/webservices/complex/statelessSession` example, using the `build.xml` Ant build file in the example directory to perform all the main steps, such as compiling the EJB Java source code into class files, executing the `servicegen` WebLogic Web Service Ant task, and deploying the Web Service to WebLogic Server. The section "Anatomy of the Example" on page 3-4 describes the main components (such as the stateless session EJB and the non-built-in data type) that make up the example.

Note: It is assumed in this section that you have started the examples WebLogic Server domain that is installed by default with WebLogic Server. The domain directory of the examples server is *`WL_HOME`*`\samples\domains\examples`.

To build and run the sample `Trader` WebLogic Web Service:

1. Open a command window.
2. Set up your environment.

   On Windows NT, execute the `setExamplesEnv.cmd` command, located in the directory *`WL_HOME`*`\samples\domains\examples`, where *`WL_HOME`* is the main directory of your WebLogic Platform installation, such as `d:\beahome\weblogic81`.

   On UNIX, execute the `setEnv.sh` command, located in the directory *`WL_HOME`*`/samples/domains/examples`, where *`WL_HOME`* is the main directory of your WebLogic Platform installation, such as `/beahome/weblogic81`.
3. Change to the *`WL_HOME`*`\samples\server\examples\src\examples\webservices\complex\statelessSession` directory.
4. Assemble and compile the example by executing the Java `ant` utility at the command line:

   ```
   prompt> ant
   ```

   The `ant` utility uses the `build.xml` build script to perform the following tasks:

   – Create an EJB JAR file from the EJB `*.java` files.

- Execute the `servicegen` Ant task which automatically generates the serialization class for the `TradeResult` non-built-in data type, creates the `web-services.xml` file, and packages all these components into a deployable EAR file.
- Deploy the EAR file on WebLogic Server.
- Execute the `clientgen` Ant task to create a local client JAR file that contains all the needed classes and interfaces to invoke the Web Service.
- Compile the `Client.java` client application used to invoke the Web Service.

5. In your development shell, run the `Client` Java application using the following command:

   ```
   prompt> ant run
   ```

   If the example runs successfully, you should see the following output in both the window from which you ran the client application and the WebLogic Server console window:

   ```
   [java] Buying 100 shares of BEAS.
   [java] Result traded 100 shares of BEAS
   [java] Buying 200 shares of MSFT.
   [java] Result traded 200 shares of MSFT
   [java] Buying 300 shares of AMZN.
   [java] Result traded 300 shares of AMZN
   [java] Buying 400 shares of HWP.
   [java] Result traded 400 shares of HWP
   [java] Selling 100 shares of BEAS.
   [java] Result traded 100 shares of BEAS
   [java] Selling 200 shares of MSFT.
   [java] Result traded 200 shares of MSFT
   [java] Selling 300 shares of AMZN.
   [java] Result traded 300 shares of AMZN
   [java] Selling 400 shares of HWP.
   [java] Result traded 400 shares of HWP
   ```

6. Optionally view the Web Service Home Page by entering the following URL in your browser:

   `http://`*`localhost`*`:`*`port`*`/webservice/TraderService`

   where

   - *`localhost`* refers to the machine on which WebLogic Server is running.
   - *`port`* refers to port on which WebLogic Server is listening.

   From the Web Service Home Page you can view the generated WSDL, and test the Web service to make sure it is working correctly.

Anatomy of the Example

This section describes the following main components of the example you built and ran in "Building and Running the Trader WebLogic Web Service Example" on page 3-2:

- "The EJB Java Interfaces and Implementation Class" on page 3-4
- "The Non-Built-In Data Type TraderResult" on page 3-8
- "The EJB Deployment Descriptors" on page 3-9
- "The servicegen Ant Task That Assembles the Web Service" on page 3-11
- "The Client Application to Invoke The Web Service" on page 3-11

The EJB Java Interfaces and Implementation Class

The Web Service example is based on a stateless session EJB. This `Trader` EJB defines two methods, `buy()` and `sell()`, that take as input a `String` stock symbol and an `int` number of shares to buy or sell. Both methods return a non-built-in data type called `TraderResult`.

The following Java interfaces and implementation class define the `Trader` EJB:

- "Remote Interface (Trader.java)" on page 3-4
- "Session Bean Implementation Class (TraderBean.java)" on page 3-5
- "Home Interface (TraderHome.java)" on page 3-8

Remote Interface (Trader.java)

```
package examples.webservices.complex.statelessSession;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

/**
 * The methods in this interface are the public face of TraderBean.
 * The signatures of the methods are identical to those of the EJBean, except
 * that these methods throw a java.rmi.RemoteException.
 * Note that the EJBean does not implement this interface. The corresponding
 * code-generated EJBObject, TraderBeanE, implements this interface and
 * delegates to the bean.
 *
 * @author Copyright (c) 1999-2003 by BEA Systems, Inc. All Rights Reserved.
 */
public interface Trader extends EJBObject {
```

```
  /**
   * Buys shares of a stock.
   *
   * @param stockSymbol       String Stock symbol
   * @param shares            int Number of shares to buy
   * @return                  TradeResult Trade Result
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   */
  public  TradeResult buy (String stockSymbol, int shares)
    throws RemoteException;

  /**
   * Sells shares of a stock.
   *
   * @param stockSymbol       String Stock symbol
   * @param shares            int Number of shares to sell
   * @return                  TradeResult Trade Result
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   */
  public TradeResult sell (String stockSymbol, int shares)
    throws RemoteException;
}
```

Session Bean Implementation Class (TraderBean.java)

```
package examples.webservices.complex.statelessSession;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 * TraderBean is a stateless Session Bean. This bean illustrates:
 * <ul>
 * <li> No persistence of state between calls to the Session Bean
 * <li> Looking up values from the Environment
 * </ul>
 *
 * @author Copyright (c) 1999-2003 by BEA Systems, Inc. All Rights Reserved.
 */
public class TraderBean implements SessionBean {
```

```
  private static final boolean VERBOSE = true;
  private SessionContext ctx;
  private int tradeLimit;

  // You might also consider using WebLogic's log service
  private void log(String s) {
    if (VERBOSE) System.out.println(s);
  }

  /**
   * This method is required by the EJB Specification,
   * but is not used by this example.
   *
   */
  public void ejbActivate() {
    log("ejbActivate called");
  }

  /**
   * This method is required by the EJB Specification,
   * but is not used by this example.
   *
   */
  public void ejbRemove() {
    log("ejbRemove called");
  }

  /**
   * This method is required by the EJB Specification,
   * but is not used by this example.
   *
   */
  public void ejbPassivate() {
    log("ejbPassivate called");
  }

  /**
   * Sets the session context.
   *
   * @param ctx               SessionContext Context for session
   */
  public void setSessionContext(SessionContext ctx) {
    log("setSessionContext called");
    this.ctx = ctx;
  }

  /**
   * This method corresponds to the create method in the home interface
   * "TraderHome.java".
   * The parameter sets of the two methods are identical. When the client calls
```

```
 * <code>TraderHome.create()</code>, the container allocates an instance of
 * the EJBean and calls <code>ejbCreate()</code>.
 *
 * @exception               javax.ejb.CreateException if there is
 *                          a communications or systems failure
 * @see                     examples.ejb11.basic.statelessSession.Trader
 */
public void ejbCreate () throws CreateException {
  log("ejbCreate called");
  try {
    InitialContext ic = new InitialContext();
    Integer tl = (Integer) ic.lookup("java:/comp/env/tradeLimit");
    tradeLimit = tl.intValue();
  } catch (NamingException ne) {
    throw new CreateException("Failed to find environment value "+ne);
  }
}

/**
 * Buys shares of a stock for a named customer.
 *
 * @param customerName      String Customer name
 * @param stockSymbol       String Stock symbol
 * @param shares            int Number of shares to buy
 * @return                  TradeResult Trade Result
 *                          if there is an error while buying the shares
 */
public TradeResult buy(String stockSymbol, int shares) {
  if (shares > tradeLimit) {
    log("Attempt to buy "+shares+" is greater than limit of "+tradeLimit);
    shares = tradeLimit;
  }
  log("Buying "+shares+" shares of "+stockSymbol);
  return new TradeResult(shares, stockSymbol);
}

/**
 * Sells shares of a stock for a named customer.
 *
 * @param customerName      String Customer name
 * @param stockSymbol       String Stock symbol
 * @param shares            int Number of shares to buy
 * @return                  TradeResult Trade Result
 *                          if there is an error while selling the shares
 */
public TradeResult sell(String stockSymbol, int shares) {
  if (shares > tradeLimit) {
    log("Attempt to sell "+shares+" is greater than limit of "+tradeLimit);
    shares = tradeLimit;
```

```
    }
    log("Selling "+shares+" shares of "+stockSymbol);
    return new TradeResult(shares, stockSymbol);
  }
}
```

Home Interface (TraderHome.java)

```
package examples.webservices.complex.statelessSession;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;

/**
 * This interface is the home interface for the TraderBean.java,
 * which in WebLogic is implemented by the code-generated container
 * class TraderBeanC. A home interface may support one or more create
 * methods, which must correspond to methods named "ejbCreate" in the EJBean.
 *
 * @author Copyright (c) 1998-2003 by BEA Systems, Inc. All Rights Reserved.
 */
public interface TraderHome extends EJBHome {
  /**
   * This method corresponds to the ejbCreate method in the bean
   * "TraderBean.java".
   * The parameter sets of the two methods are identical. When the client calls
   * <code>TraderHome.create()</code>, the container
   * allocates an instance of the EJBean and calls <code>ejbCreate()</code>.
   *
   * @return                  Trader
   * @exception               RemoteException if there is
   *                          a communications or systems failure
   * @exception               CreateException
   *                          if there is a problem creating the bean
   * @see                     examples.ejb11.basic.statelessSession.TraderBean
   */
  Trader create() throws CreateException, RemoteException;
}
```

The Non-Built-In Data Type TraderResult

The two methods of the EJB return a non-built-in data type called `TraderResult`. The following Java code describes this data type:

```
package examples.webservices.complex.statelessSession;
```

```
import java.io.Serializable;

/**
 * This class reflects the results of a buy/sell transaction.
 *
 * @author Copyright (c) 1999-2003 by BEA Systems, Inc. All Rights Reserved.
 */
public final class TradeResult implements Serializable {

  // Number of shares really bought or sold.
  private int    numberTraded;

  private String stockSymbol;

  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }

  public void setNumberTraded(int numberTraded) {
        this.numberTraded = numberTraded;
  }

  public String getStockSymbol() { return stockSymbol; }

  public void setStockSymbol(String stockSymbol) {
        this.stockSymbol = stockSymbol;
  }
}
```

The EJB Deployment Descriptors

The `Trader` EJB defines the following two deployment descriptor files to describe itself:

- "ejb-jar.xml" on page 3-9
- "weblogic-ejb-jar.xml" on page 3-10

ejb-jar.xml

```
<?xml version="1.0"?>

<!DOCTYPE ejb-jar PUBLIC
'-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN'
'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>
```

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>TraderService</ejb-name>
      <home>examples.webservices.complex.statelessSession.TraderHome</home>
      <remote>examples.webservices.complex.statelessSession.Trader</remote>

<ejb-class>examples.webservices.complex.statelessSession.TraderBean</ejb-class
>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <env-entry>
        <env-entry-name>WEBL</env-entry-name>
        <env-entry-type>java.lang.Double </env-entry-type>
        <env-entry-value>10.0</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>INTL</env-entry-name>
        <env-entry-type>java.lang.Double </env-entry-type>
        <env-entry-value>15.0</env-entry-value>
      </env-entry>
      <env-entry>
        <env-entry-name>tradeLimit</env-entry-name>
        <env-entry-type>java.lang.Integer </env-entry-type>
        <env-entry-value>500</env-entry-value>
      </env-entry>
    </session>
  </enterprise-beans>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>TraderService</ejb-name>
        <method-name>*</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

weblogic-ejb-jar.xml

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC
'-//BEA Systems, Inc.//DTD WebLogic 7.0.0 EJB//EN'
'http://www.bea.com/servers/wls700/dtd/weblogic700-ejb-jar.dtd'>

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
```

```
    <ejb-name>TraderService</ejb-name>
    <jndi-name>webservices-complex-statelessession</jndi-name>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```

The servicegen Ant Task That Assembles the Web Service

The Ant build file, `build.xml`,contains a call to the `servicegen` Ant task that introspects the `trader.jar` EJB file and:

- Generates all data type components (such as the serialization class).
- Creates the `web-services.xml` deployment descriptor file.
- Packages all components into a deployable `trader.ear` file.

The following snippet from the sample `build.xml` file contains instructions that will build the EAR file into a temporary `build_dir` directory:

```
<target name="build" >
    <delete dir="build_dir" />
    <mkdir dir="build_dir" />
    <copy todir="build_dir" file="trader.jar"/>
    <servicegen
      destEar="build_dir/trader.ear"
      warName="trader.war"
      contextURI="webservice">
      <service
         ejbJar="build_dir/trader.jar"
         targetNamespace="http://www.bea.com/examples/Trader"
         serviceName="TraderService"
         serviceURI="/TraderService"
         generateTypes="True"
         expandMethods="True" >
      </service>
    </servicegen>
  </target>
```

The Client Application to Invoke The Web Service

The following Java client application shows how to use the JAX-RPC API to invoke the buy and sell operations of the deployed `Trader` Web Service:

```
package examples.webservices.complex.statelessSession;
```

```
/**
 * This class illustrates how to use the JAX-RPC API to invoke the TraderService
 * Web service to perform the following tasks:
 * <ul>
 * <li> Buy 100 shares of some stocks
 * <li> Sell 100 shares of some stocks
 * </ul>
 *
 * The TraderService Web service is implemented using the Trader
 * stateless session EJB.
 * * @author Copyright (c) 1998-2003 by BEA Systems, Inc. All Rights Reserved.
 */

public class Client {

  public static void main(String[] args) throws Exception {

    // Setup the global JAXM message factory
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // Parse the argument list
    Client client = new Client();
    String wsdl = (args.length > 0? args[0] : null);
    client.example(wsdl);
  }

  public void example(String wsdlURI) throws Exception {

    TraderServicePort trader = null;
    if (wsdlURI == null) {
      trader = new TraderService_Impl().getTraderServicePort();
    } else {
      trader = new TraderService_Impl(wsdlURI).getTraderServicePort();
    }
    String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };

      // execute some buys
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Buying "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.buy(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());
    }
```

```
    // execute some sells
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Selling "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.sell(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());

    }

  }

  private static void log(String s) {
    System.out.println(s);
  }

}
```

CHAPTER 4

Designing WebLogic Web Services

The following sections discuss design considerations for implementing WebLogic Web Services:

- "Choosing the Back-end Components of Your Web Service" on page 4-1
- "Choosing Between Synchronous or Asynchronous Operations" on page 4-2
- "Choosing RPC-Oriented or Document-Oriented Web Services" on page 4-3
- "Using Built-In and Non-Built-In Data Types" on page 4-4
- "Using SOAP Message Handlers to Intercept the SOAP Message" on page 4-5
- "Mimicking a Conversational (Stateful) WebLogic Web Service" on page 4-5

Choosing the Back-end Components of Your Web Service

You implement a WebLogic Web Service operation with one of the following types of back-end component:

- A method of a stateless session EJB
- A method of a Java class
- A JMS message consumer or producer.

Note: BEA recommends that you implement your Web Service operation with only a stateless session EJB or a Java class, and not with a JMS consumer or producer. In most of the book, it is assumed that your Web Service is implemented with either an

EJB or a Java class. All JMS-specific information is in its own chapter: Chapter 16, "Creating JMS-Implemented WebLogic Web Services."

EJB Back-end Component

Web Service operations implemented with a method of a stateless session EJB are interface driven, which means that the business methods of the underlying stateless session EJB determine how the Web Service operation works. When clients invoke the Web Service operation, they send parameter values to the method, which executes and sends back the return value.

Use a stateless session EJB back-end component if your Web Service will have the following characteristics:

- The behavior of the Web Service can be expressed as an interface.
- The Web Service is process-oriented rather than data-oriented.
- The Web Service can benefit from EJB facilities, such as persistence, security, transactions, and concurrency.

Examples of this Web Service operation implementation include providing the current weather conditions in a particular location; returning the current price for a given stock; or checking the credit rating of a potential trading partner prior to the completion of a business transaction.

Java Class Back-end Component

Web Service operations implemented with Java classes are similar to those implemented with an EJB method. Creating a Java class, however, is often simpler and faster than creating an EJB. Use a Java class as a back-end component when you do not need the overhead of EJB facilities such as persistence, security, transactions, and concurrency.

There are limitations and restrictions to using a Java class as a back-end component, however. For details, see "Implementing a Web Service By Writing a Java Class" on page 5-4.

Choosing Between Synchronous or Asynchronous Operations

WebLogic Web Service operations can be either synchronous request-response or asynchronous one-way.

Synchronous request-response (the default behavior) means that every time a client application invokes a Web Service operation, it receives a SOAP response, even if the method that

implements the operation returns `void`. Asynchronous one-way means that the client never receives a SOAP response, even a fault or exception.

You specify this type of behavior with the `invocation-style` attribute of the `<operation>` element in the `web-services.xml` file.

Web Service operations are typically synchronous request-response, mirroring typical RPC-style behavior. Sometimes, however, you might want to implement asynchronous behavior if your client application has no need for a response, even in the case of an error. When designing asynchronous one-way Web Service operations, keep the following issues in mind:

- The back-end component that implements the operation *must* explicitly return `void`.
- You cannot specify out or in-out parameters to the operation, you can only specify in parameters.

Choosing RPC-Oriented or Document-Oriented Web Services

The operations of a WebLogic Web Service can be either RPC-oriented or document-oriented. As described in the WSDL 1.1 specification, an RPC-oriented operation is one in which the SOAP messages contain parameters and return values, and a document-oriented operation is one in which the SOAP messages contain XML documents.

WebLogic Server supports two types of document-oriented Web Service operations: standard document and document-wrapped.

Standard document-oriented Web Service operations take only one parameter, typically an XML document. This means that the methods that implement the operations must also have only one parameter. Document-wrapped Web Service operations, however, can take any number of parameters, although the parameter values will be wrapped into one complex data type in the SOAP messages. If two or more methods of your stateless session EJB or Java class that implement the Web Service have the same number and data type of parameters, and you want the operations to be document-oriented, you must specify that they be document-wrapped.

There are no restrictions on the number of parameters of an RPC-oriented operation.

RPC-oriented WebLogic Web Service operations use SOAP encoding. Document-oriented WebLogic Web Service operations use literal encoding.

All operations in a single WebLogic Web Service must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web Service.

By default, the operations of a WebLogic Web Service are RPC-oriented. If you want to specify that the operations are document-oriented, use the `style="document"` or `style="documentwrapped"` attribute of the `<service>` element when assembling a Web Service using the `servicegen` Ant task. The generated `web-services.xml` deployment descriptor will contain a corresponding `style="document"` or `style="documentwrapped"` attribute for the appropriate `<web-service>` element.

For information on implementing document-oriented WebLogic Web Services, see "Implementing a Document-Oriented Web Service" on page 5-6. For details on using the `servicegen` Ant task to assemble a document-oriented Web Service, see "Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3 and "servicegen" on page B-25.

Using Built-In and Non-Built-In Data Types

WebLogic Web Services support both built-in and non-built-in data types as parameters and return values to Web Services operations. This means that WebLogic Web Services can handle any type of data that can be represented using XML Schema.

Built-in data types are those specified by the JAX-RPC specification. If your Web Service uses only built-in data types, the conversion of the data between its XML and Java representation is handled automatically by WebLogic Server. For the full list of built-in data types, see "Supported Built-In Data Types" on page 5-15.

If, however, your Web Service operation is more complex and uses a non-built-in data type as a parameter or return value, you must:

a. Create the serialization class that convert the data between its XML and Java representation

b. Describe the XML representation of the data type (using XML Schema notation) in the `web-services.xml` file

c. Create the Java class file of the data type

d. Describe the data type mapping in the `web-services.xml` file

WebLogic Server includes Ant tasks that you use to perform these tasks for many common XML and Java data types; this feature is called *autotyping*. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16. For information on running these Ant tasks, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

Note: If you are using the autotyping Ant tasks to generate data type information for a Java class, your class must conform to the guidelines described in "Implementing Non-Built-In Data Types" on page 5-5.

If your data type is not either built-in or one of the supported non-built-in data types, then you must create the serialization class, and so on, manually. For details, see Chapter 11, "Using Non-Built-In Data Types."

Using SOAP Message Handlers to Intercept the SOAP Message

Some Web Services need access to the SOAP message, for which you can create SOAP message handlers.

A SOAP message handler provides a mechanism for intercepting the SOAP message in both the request and response of the Web Service. You can create handlers in both the Web Service itself and the client applications that invoke the Web Service.

A simple example of using handlers is to encrypt and decrypt secure data in the body of a SOAP message. A client application uses a handler to encrypt the data before it sends the SOAP message request to the Web Service. The Web Service receives the request and uses a handler to decrypt the data before it sends the data to the back-end component that implements the Web Service. The same steps happen in reverse for the response SOAP message.

Another example is accessing information in the header part of the SOAP message. You can use the SOAP header to store Web Service specific information and then use handlers to manipulate it.

You can also use SOAP message handlers to improve the performance of your Web Service. After your Web Service has been deployed for a while, you might discover that many consumers invoke it with the same parameters. You could improve the performance of your Web Service by caching the results of popular invokes of the Web Service (assuming the results are static) and immediately returning these results when appropriate, without ever invoking the back-end components that implement the Web Service. You implement this performance improvement by using handlers to check the request SOAP message to see if it contains the popular parameters.

Mimicking a Conversational (Stateful) WebLogic Web Service

You implement a WebLogic Web Service operation using stateless session EJBs or Java classes, and thus a WebLogic Web Service operation is not stateful, or one that can conduct a back and forth conversation beyond the standard request/response model.

You can, however, mimic a conversational Web Service by using JDBC or entity beans. For example, you could design a Web Service so that client applications that invoke it pass a unique ID to identify themselves to the stateless session EJB entry point. This EJB uses the ID to persist the conversation in persistent storage, using either entity beans or JDBC. The next time the same client application invokes the Web Service, the stateless session EJB can recover the previous state of the conversation by selecting the persisted data using the unique ID.

For information on programming entity beans, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html. For information on JDBC, see WebLogic jDrivers at http://e-docs.bea.com/wls/docs81/jdrivers.html.

CHAPTER 5

Implementing WebLogic Web Services

The following sections describe how to implement WebLogic Web Services:

- "Overview of Implementing a WebLogic Web Service" on page 5-1
- "Examples of Implementing WebLogic Web Services" on page 5-2
- "Implementing a WebLogic Web Service: Main Steps" on page 5-2
- "Writing the Java Code for the Components" on page 5-3
- "Supported Built-In Data Types" on page 5-15

Overview of Implementing a WebLogic Web Service

Implementing a WebLogic Web Service refers to writing and compiling the Java code for the back-end components that make up the Web Service and, if necessary, creating SOAP message handlers and Java code for the non-built-in data types. Back-end components include stateless session EJBs and Java classes. A Web Service can be implemented with multiple combinations of these components.

A single WebLogic Web Service consists of one or more operations; you can implement each operation using methods of different back-end components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

If you are implementing a WebLogic Web Service from an existing WSDL file, you can use the WebLogic Server `wsdl2Service` Ant task to automatically generate the Java interface that represents your Web Service, optionally generate an empty implementation Java class file, then

write the business-logic code for the Java implementation class to make the Web Service behave as you want.

It is assumed that you have read and understood the design issues discussed in Chapter 4, "Designing WebLogic Web Services," designed your Web Service and that you know the types of components you need to create.

Note: BEA recommends that you implement your Web Service operation with only a stateless session EJB or a Java class, and not with a JMS consumer or producer. In most of the book, it is assumed that your Web Service is implemented with either an EJB or a Java class. All JMS-specific information is in its own chapter: Chapter 16, "Creating JMS-Implemented WebLogic Web Services."

Examples of Implementing WebLogic Web Services

WebLogic Server includes examples of implementing WebLogic Web Services in the *WL_HOME*`/samples/server/examples/src/examples/webservices` directory, where *WL_HOME* refers to the main WebLogic Platform directory. For detailed instructions on how to build and run the examples, open the following Web page in your browser:

WL_HOME`/samples/server/examples/src/examples/webservices/package-summary.html`

Implementing a WebLogic Web Service: Main Steps

The following procedure describes the high-level steps to implement a WebLogic Web Service. Later parts of this document describe the steps in more detail. Although some of the steps are mandatory, others are optional, depending on the type of Web Service you are implementing.

1. Write the Java code for the back-end components that make up the Web Service.

   See "Writing the Java Code for the Components" on page 5-3.

2. If you need to process information in the SOAP request or response or directly access the SOAP attachments, create SOAP message handlers and handler chains.

   See Chapter 12, "Creating SOAP Message Handlers to Intercept the SOAP Message."

3. If your back-end components use non-built-in data types as parameters or return values, generate or create the Java code for the data type as well as the serialization class that converts the data between XML and Java.

   See "Implementing Non-Built-In Data Types" on page 5-5.

4. Compile the Java code into class files. For details, see *Compiling Java Code* at http://e-docs.bea.com/wls/docs81/programming/topics.html#Compiling.

Writing the Java Code for the Components

When you implement a WebLogic Web Service, you write Java code for one of these back-end components:

- A stateless session EJB.

  See "Implementing a Web Service By Writing a Stateless Session EJB" on page 5-4 for information on writing the Java code. For an example, see "The EJB Java Interfaces and Implementation Class" on page 3-4.

- A Java class.

  See "Implementing a Web Service By Writing a Java Class" on page 5-4 for information on writing the Java code. For an example, see the *WL_HOME*`/samples/server/examples/src/examples/webservices/basic/javaclass` directory, where *WL_HOME* refers to the main WebLogic Platform installation directory:

If your Web Service operations use non-built-in data types as parameters or return values, see "Implementing Non-Built-In Data Types" on page 5-5.

If you are implementing a Web Service that uses document-oriented operations, rather than the default RPC-oriented, see "Implementing a Document-Oriented Web Service" on page 5-6.

If you are implementing a WebLogic Web Service based on an existing WSDL file, and you want to implement the Web Service with a Java class, use the WebLogic Server `wsdl2Service` Ant task to generate the Web Service interface, and optional implementation, class to use as a starting point. For details about using this Ant task, see "Generating a Partial Implementation From a WSDL File" on page 5-6.

For information about using SOAP Attachments, see "Using SOAP Attachments" on page 5-9.

For information on throwing exceptions from your Web Service implementation, see "Throwing SOAP Fault Exceptions" on page 5-14.

If you want your Web Service operation to return multiple values, see "Implementing Multiple Return Values" on page 5-10.

Implementing a Web Service By Writing a Stateless Session EJB

Writing the Java code for the stateless session EJB for a Web Service is no different from writing a stand-alone EJB, with these exceptions:

- You can specify in the `web-services.xml` deployment descriptor that a Web Service operation is *one-way*, which means that the client application that invokes the Web Service does not wait for a response. When you write the Java code for the EJB method that implements this type of operation, you *must* specify that it return `void`.

  For more information on specifying in the `web-services.xml` file that a Web Service operation is one-way, see operation.

- If the data type of the parameters or return value of an EJB method are not part of the set of built-in data types, then you must generate or create the serialization class that converts these data types between their XML and Java representations. For the list of built-in data types, see "Supported Built-In Data Types" on page 5-15.

  See "Implementing Non-Built-In Data Types" on page 5-5.

For an example of how to write a stateless session EJB, see "The EJB Java Interfaces and Implementation Class" on page 3-4. For general information, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html.

Implementing a Web Service By Writing a Java Class

You can implement a Web Service operation using a Java class as long as you follow these rules:

- Do not start any threads. This rule applies to all Java code that runs on WebLogic Server.
- Define a default no-argument constructor.
- Define as public the methods of the Java class that are going to be exposed as Web Service operations.
- Write thread-safe Java code, because WebLogic Server maintains only a single instance of a Java class that implements a Web Service operation, and each invoke of the Web Service uses this same instance.

Although it is not required, your Java class can extend the JAX-RPC `javax.xml.rpc.server.ServiceLifecycle` interface, which defines the life cycle for the Web Service endpoint. However, because this version of WebLogic Server does not support servlets as back-end components of WebLogic Web Services, BEA does *not* provide an

implementation of the `javax.xml.rpc.server.ServletEndpointContext` interface. This means that if your Java class extends the `ServiceLifecycle` interface, its `init()` method is passed `null` rather than an instance of `ServletEndpointContext`.

For an example of implementing a WebLogic Web Service operation with a Java class, go to the *`WL_HOME`*`/samples/server/examples/src/examples/webservices/basic/javaclass` directory, where *`WL_HOME`* refers to the main directory of your WebLogic Server installation.

Implementing Non-Built-In Data Types

Stateless session EJBs and Java classes do not necessarily take built-in data types as parameters and return values, but rather, might use a Java data type that you create yourself. An example of a non-built-in data type is `TradeResult`, which has two fields: a String stock symbol and an integer number of shares traded. For the list of built-in data types, see "Supported Built-In Data Types" on page 5-15.

If your back-end components use non-built-in data types as parameters or return values, you must create the Java code of the data type and then create or generate the serialization class that converts the data between XML and Java.You can do this in one of two ways:

- Use WebLogic Server `servicegen` or `autotype` Ant tasks to introspect your EJB and automatically generate the serialization class. These Ant tasks also create the corresponding XML Schema to represent your data in XML format and update your `web-services.xml` deployment descriptor file with the relevant data type mapping information. You will run these Ant tasks as part of assembling the Web Service, described in "Creating the Build File That Specifies the servicegen Ant Task" on page 6-4 and "Running the autotype Ant Task" on page 6-9.

  Warning: The serialization class and Java and XML representations generated by the `autotype` and `servicegen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

- Create the serialization class and XML and Java representations of your data type manually. This method is more complex and time-consuming than generating them using the Ant task. For details on handling non-built-in data types manually, see Chapter 11, "Using Non-Built-In Data Types."

If you are going to create the XML representation of your Java data type manually, along with the serialization class, you can code the Java class any way you want, because you will be writing all the conversion code yourself.

If you are going to use the `servicegen` or `autotype` Ant tasks to generate the data type components automatically, follow these requirements when writing the Java class for your data type:

- Define a default constructor, which is a constructor that takes no parameters.
- Define both `getXXX()` and `setXXX()` methods for each member variable that you want to expose.
- Make the data type of each exposed member variable one of the built-in data types, or a non-built-in data type that consists of built-in data types and has the corresponding serialization class and XML Schema representation.

The `servicegen` and `autotype` Ant tasks can generate data type components for most common XML and Java data types. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.

Implementing a Document-Oriented Web Service

WebLogic Web Services can be either document-oriented (the SOAP message contains a document) or RPC-oriented (the SOAP message contains parameters and return values). By default, WebLogic Web Services are RPC-oriented. You specify that a Web Service is document-oriented when you assemble it using the `servicegen` Ant task.

The procedures in this chapter assume that you are creating an RPC-oriented Web Service. If, however, you are creating a document-oriented Web Service, follow these additional guidelines when implementing the back-end component:

- The methods that implement each operation of the Web Service can have only *one* parameter. This single parameter can be of any supported data type; see "Using Built-In and Non-Built-In Data Types" on page 4-4 for more information.
- The methods that implement each operation cannot use out and in-out parameters.

Generating a Partial Implementation From a WSDL File

It is assumed in most of this chapter that you are implementing a Web Service by writing the back-end component first. Sometimes, however, you might need to start with an existing WSDL from which you create the implementation. For example, your company might include a corporate architecture group that defines common service descriptions, specifically WSDL files, that must be implemented by different departments. In this case you can use the `wsdl2Service` Ant task to generate a partial implementation.

The `wsdl2Service` Ant task takes as input an existing WSDL file and generates:

- the Java interface that represents the implementation of your Web Service
- optionally, an empty Java implementation class
- the `web-services.xml` file that describes the Web Service

The generated Java interface file describes the template for the full Java class-implemented WebLogic Web Service. The template includes full method signatures that correspond to the operations in the WSDL file. You must then write a Java class that implements this interface so that the methods function as you want, following the guidelines in "Implementing a Web Service By Writing a Java Class" on page 5-4. You can generate a skeleton of the implementation class by specifying the `generateImpl="True"` attribute; add the business logic Java code to this class to complete the implementation.

The `wsdl2Service` Ant task generates a Java interface for only one Web Service in a WSDL file (specified by the `<service>` element.) Use the `serviceName` attribute to specify a particular service; if you do not specify this attribute, the `wsdl2Service` Ant task generates a Java interface for the first `<service>` element in the WSDL.

Warning: The `wsdl2Service` Ant task, when generating the `web-services.xml` file for your Web Service, assumes you use the following convention when naming the Java class that implements the generated Java interface:

`packageName.serviceName``Impl`

where *`packageName`* and *`serviceName`* are the values of the similarly-named attributes of the `wsdl2Service` Ant task. The Ant task puts this information in the `class-name` attribute of the `<java-class>` element of the `web-services.xml` file.

If you name your Java implementation class differently, you must manually update the generated `web-services.xml` file accordingly.

Running the wsdl2Service Ant Task

To run the `wsdl2Service` Ant task, follow these steps:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the `wsdl2Service` Ant task. For details, see "Sample build.xml Files for the wsdl2Service Ant Task" on page 5-8.
3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

4. Create a Java class implementation for the interface generated in the `myService/implementation` directory. For details, see "Implementing a Web Service By Writing a Java Class" on page 5-4.

For reference information about the `wsdl2Service` Ant task, see "wsdl2Service" on page B-46.

Sample build.xml Files for the wsdl2Service Ant Task

The following example shows a simple `build.xml` file:

```
<project name="buildWebservice" default="generate-from-WSDL">
   <target name="generate-from-WSDL">
      <wsdl2service
           wsdl="wsdls/myService.wsdl"
           destDir="myService/implementation"
           typeMappingFile="autotype/types.xml"
           packageName="example.ws2j.service" />
   </target>
</project>
```

In the example, the `wsdl2Service` Ant task generates a Java interface file for the first `<service>` element it finds in the WSDL file `wsdls/myService.wsdl`. It uses data type mapping information for any non-built-in data types from the `autotype/types.xml` file; typically you have previously run the `autotype` Ant task to generate this file. The Java interface file and `web-services.xml` file are generated into the directory `myService/implementation`.

Assume that value of the `name` attribute of the first `<service>` element in the WSDL file is `SuperDooperService`. The `wsdl2Service` generates a Java interface called `example.ws2j.service.SuperDooperService` and assumes that your Java implementation class will be `example.ws2j.service.SuperDooperServiceImpl`.

Using SOAP Attachments

Certain Java data types, if used as parameters or return values of a method that implements a Web Service operation, are automatically transported as SOAP Attachments (rather than elements in the SOAP body) when going over the wire. The following table lists the Java data types and their corresponding MIME type in the SOAP Attachment.

Table 5-1 Mapping Java Data Types to MIME Types

Java Data Type	MIME Type
`java.awt.Image`	`image/gif` or `image/jpeg`
`java.lang.String`	`text/plain`
`javax.mail.internet.MimeMultipart`	`multipart/*`
`javax.xml.transform.Source`	`text/xml` or `application/xml`
`javax.activation.DataHandler`	Depends on the data represented in specific instance of the `DataHandler`.

If you code the method of the Java class or EJB to use one of the preceding Java data types as a parameter or return value and then use `servicegen` to assemble the Web Service, the generated `web-services.xml` deployment descriptor file automatically specifies that the actual data of the parameter or return value is in the SOAP attachment. In particular, the `location` attribute of the `<param>` or `<return-param>` is set equal to `attachment`. When a client application invokes the Web Service, the WebLogic Web Services runtime automatically looks for the parameter data in the attachment to the SOAP request, or adds the return value data to the attachment in the SOAP response, depending on whether it is the parameter or return value or both that is one of the Java data types listed in the preceding table.

If you want to access and manipulate the SOAP attachment directly, you must create a SOAP message handler and use the SOAP with Attachments API for Java 1.1 (SAAJ). For details, see Chapter 12, “Creating SOAP Message Handlers to Intercept the SOAP Message.”

java.lang.String

The Java data type `java.lang.String` works a little differently than what is described in the preceding section. By default, if you use `java.lang.String` in the method that implements a Web Service operation, the `servicegen` Ant tasks and the WebLogic Web Services runtime treat it as a built-in data type. This means that the data will be part of the SOAP body as an XML Schema `string` type rather than a `text/plain` MIME type in a SOAP attachment.

If, however, you want the `java.lang.String` parameter or return value to be transported as a `text/plain` MIME encoded SOAP attachment, you must manually update the `web-services.xml` deployment descriptor file and change the value of the `location` attribute of the corresponding `<param>` or `<return-value>` element from the default `Body` to `attachment`.

For more information on the attributes and elements of the `web-services.xml` file, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

javax.activation.DataHandler

You use the `javax.activation.DataHandler` data type to represent data in a SOAP attachment that is not listed in the table in "Using SOAP Attachments" on page 5-9.

The `DataHandler` class provides a consistent interface to data available in many different sources and formats. It manages simple stream to string conversions and related operations using `javax.activation.DataContentHandlers`. The `DataHandler` class also provides access to commands that can operate on the data. The commands are found using a `javax.activation.CommandMap` class.

`DataHandlers` are part of the J2EE JavaBeans Activation Framework standard extension. It is assumed that if you use a `DataHandler` as a parameter or return type of a WebLogic Web Service operation, you have implemented all the needed components, such as the `DataContentHandler`.

For general information about `DataHandlers` and how to implement them, see JavaBeans Activation Framework at http://java.sun.com/products/javabeans/glasgow/jaf.html. See the Javadoc at http://java.sun.com/products/javabeans/glasgow/javadocs/index.html for a description of the JavaBeans Activation Framework APIs.

Implementing Multiple Return Values

WebLogic Web Service operations typically return a single value: the return value of the EJB or Java class method that implements the Web Service operation. If you want a Web Service operation to return multiple values, you can either:

- Define the data type of the return value to be a complex non-built-in type, such as an object with multiple parts or an array. For details about implementing these data types, see "Implementing Non-Built-In Data Types" on page 5-5.
- Specify that one or more of the parameters of the Web Service operation be *out* or *in-out* parameters. This section describes how to create these types of parameters.

Using Out and In-Out Parameters

Out and in-out parameters are a mechanism whereby parameters to an operation can act as both standard in parameters *and* return values. Out parameters are undefined when the operation is invoked, but *are* defined by the method that implements the operation when the operation completes. In-out parameters are defined when invoked *and* when completed. For example, assume a Web Service operation contains one out parameter, and the operation is implemented with an EJB method. The EJB method sets the value of the out parameter and sends this value back to the client application that invoked it. The client application can then access the value of this out parameter as if it were a return value. An in-out parameter is one that acts as both a standard input parameter for sending information to the method and an out parameter. This section discusses how to implement a Web Service operation with an EJB or Java class method that uses out or in-out parameters.

The following example shows a method whose second parameter is an in-out parameter:

```
public String myMethod( String param1,
        javax.xml.rpc.holders.IntHolder intHolder ) {

System.out.println ("The input value is: " + intHolder.value );
intHolder.value = 20;  // the new value of the out parameter

return param1;
}
```

You invoke the method with two parameters, a String and an integer. The method returns two values: a String (the standard return value) and an integer (via the `IntHolder` holder parameter).

Out and in-out parameters must implement the `javax.xml.rpc.holders.Holder` interface. Use the `Holder.value` field to first access the input value of an in-out parameter and then set the value of out and in-out parameters. In the preceding example, assume the method was invoked with a value of 40 as the second parameter; when the method completes, the value of `intHolder` is now 20.

Using Holder Classes to Implement Multiple Return Values

If the out or in-out parameter is a standard data type, use one of the JAX-RPC `Holder` classes, listed in the following table.

Table 5-2 Built-In Holder Classes Provided by WebLogic Server

Built-In Holder Class	Java Data Type That It Holds
`javax.xml.rpc.holders.BooleanHolder`	`boolean`
`javax.xml.rpc.holders.ByteHolder`	`byte`
`javax.xml.rpc.holders.ShortHolder`	`short`
`javax.xml.rpc.holders.IntHolder`	`int`
`javax.xml.rpc.holders.LongHolder`	`long`
`javax.xml.rpc.holders.FloatHolder`	`float`
`javax.xml.rpc.holders.DoubleHolder`	`double`
`javax.xml.rpc.holders.BigDecimalHolder`	`java.math.BigDecimal`
`javax.xml.rpc.holders.BigIntegerHolder`	`java.math.BigInteger`
`javax.xml.rpc.holders.ByteArrayHolder`	`byte[]`
`javax.xml.rpc.holders.CalendarHolder`	`java.util.Calendar`
`javax.xml.rpc.holders.QnameHolder`	`javax.xml.namespace.QName`
`javax.xml.rpc.holders.StringHolder`	`java.lang.String`

If, however, the data type of the parameter is not provided, you must create your own implementation.

To create your own implementation of the `javax.xml.rpc.holders.Holder` interface, follow these guidelines:

- Name your implementation class *Type*`Holder`, where *Type* is the name of the complex type. For example, if your complex type is called `Person`, then your implementation class is called `PersonHolder`.

- The `Holder` implementation class should be packaged in a `holders` sub-package below the package of the class it is holding.

  For example, if your complex type `Person` is in the `examples.webservices` package, then the `PersonHolder` implementation class should be in the `examples.webservices.holders` package.

- Create a public field called `value`, whose data type is the same as that of the parameter.
- Create a default constructor that initializes the `value` field to a default value.
- Create a constructor that sets the `value` field to the value of the passed parameter.

The following example shows the outline of a `PersonHolder` implementation class:

```
package examples.webservices.holders;

public final class PersonHolder implements
            javax.xml.rpc.holders.Holder {

   public Person value;

   public PersonHolder() {

       // set the value variable to a default value
   }

   public PersonHolder (Person value) {

       // set the value variable to the passed in value
   }

}
```

After you have created the Holder implementation class for your out or in-out parameter, update the Java code for the method that implements your Web Service operation to use this Holder class. When you later use the `servicegen` Ant task to assemble your Web Service, the generated `web-services.xml` file will automatically specify that the parameter is an in-out parameter, as shown in the following excerpt:

```
<param name="inoutparam" style="inout"
       type="xsd:Person" />
```

If you want the parameter to be an out, rather than in-out, parameter, you must update the generated `web-services.xml` file manually.

For details about writing a client application that invokes a Web Services that uses out or in-out parameters, see "Writing a Client That Uses Out or In-Out Parameters" on page 7-10.

Throwing SOAP Fault Exceptions

When you write the error-handling Java code for the EJB or Java class that implements your WebLogic Web Service, you can either throw your own exceptions or throw a `javax.xml.rpc.soap.SOAPFaultException` exception. If you throw a `SOAPFaultException`, WebLogic Server maps it to a SOAP fault and sends it to the client application that invokes the operation.

If your EJB or Java class throws any other type of Java exception, WebLogic Server tries to map it to a SOAP fault as best it can. However, if you want to control what the client application receives and send it the best possible exception information, you should explicitly throw a `SOAPFaultException` exception or one that extends the exception.

The following excerpt describes the `SOAPFaultException` class:

```
public class SOAPFaultException extends java.lang.RuntimeException {
    public SOAPFaultException (QName faultcode,
                               String faultstring,
                               String faultactor,
                               javax.xml.soap.Detail detail ) {...}
    public Qname getFaultCode() {...}
    public String getFaultString() {...}
    public String getFaultActor() {...}
    public javax.xml.soap.Detail getDetail() {...}
}
```

Use the SOAP with Attachments API for Java 1.1 (SAAJ) `javax.xml.soap.SOAPFactory.createDetail()` method to create the `Detail` object, which is a container for `DetailEntry` objects that provide detailed application-specific information about the error.

The following class shows an example of creating and throwing a `SOAPFaultException` from within the implementation of your Web Service:

```
import javax.xml.soap.SOAPFactory;
import javax.xml.soap.Detail;
import javax.xml.soap.SOAPException;

import javax.xml.namespace.QName;
import javax.xml.rpc.soap.SOAPFaultException;

public class HelloWorldService{

  public void helloSOAPFault(){

    Detail detail = null;
```

```
    try{
      detail = SOAPFactory.newInstance().createDetail();
      detail.addChildElement( "MyDetails" ).addTextNode( "failed" );
    }catch( SOAPException e ){
      e.printStackTrace();
    }

    throw new SOAPFaultException(
        new QName( "http://tutorial/sample9/fault", "ServerFailed" ),
        "helloSOAPFault method failed",
        "http://foo/bar/baz/",
        detail );
  }

  public void helloCustomFault() throws HelloWorldException{
    throw new HelloWorldException( "This is my error message, " +
        "client should get this" );
  }
}
```

Warning: If you create and throw your own exception (rather than use `SOAPFaultException`) and two or more of the properties of your exception class are of the same data type, then you *must* also create setter methods for these properties, even though the JAX-RPC specification does not require it. This is because when a WebLogic Web Service receives the exception in a SOAP message and converts the XML into the Java exception class, there is no way of knowing which XML element maps to which class property without the corresponding setter methods.

Supported Built-In Data Types

The following sections describe the built-in data types supported by WebLogic Web Services and the mapping between their XML and Java representations. As long as the data types of the parameters and return values of the back-end components that implement your Web Service are in the set of built-in data types, WebLogic Server automatically converts the data between XML and Java.

If, however, you use non-built-in data types, then you must create the serialization class to convert the data between XML and Java. WebLogic Server includes the `servicegen` and `autotype` Ant tasks that can generate the serialization class for most non-built-in data types. See "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16 for a list of supported XML and Java data types. For more information about using `servicegen` and `autotype`, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

If your data type is not supported, then you must create your serialization class manually. For details, see Chapter 11, "Using Non-Built-In Data Types."

XML Schema-to-Java Mapping for Built-In Data Types

The following table lists the defined mappings for all built-in data types defined by XML Schema (target namespace `http://www.w3.org/2001/XMLSchema`) and the corresponding SOAP data types (target namespace `http://schemas.xmlsoap.org/soap/encoding/`).

For a list of the supported non-built-in XML data types, see "Supported XML Non-Built-In Data Types" on page 6-17.

Table 5-3 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
boolean	boolean
byte	byte
short	short
int	int
long	long
float	float
double	double
integer	java.math.BigInteger
decimal	java.math.BigDecimal
string	java.lang.String
dateTime	java.util.Calendar
base64Binary	byte[]
hexBinary	byte[]
duration	weblogic.xml.schema.binding.util.Duration
time	java.util.Calendar
date	java.util.Calendar

Table 5-3 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
gYearMonth	java.util.Calendar The `java.util.Calendar` Java data type contains more fields than the `gYearMonth` data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gYear	java.util.Calendar The `java.util.Calendar` Java data type contains more fields than the `gYearMonth` data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gMonthDay	java.util.Calendar The `java.util.Calendar` Java data type contains more fields than the `gYearMonth` data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gDay	java.util.Calendar The `java.util.Calendar` Java data type contains more fields than the `gYearMonth` data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
gMonth	java.util.Calendar The `java.util.Calendar` Java data type contains more fields than the `gYearMonth` data type. This additional information is not meaningful and is not generated from the actual XML data, but rather created by the data binding facility.
anyURI	java.lang.String
NOTATION	java.lang.String
token	java.lang.String
normalizedString	java.lang.String
language	java.lang.String

Table 5-3 Mapping XML Schema Data Types to Java Data Types

XML Schema Data Type	Equivalent Java Data Type (lower case indicates a primitive data type)
Name	java.lang.String
NMTOKEN	java.lang.String
NCName	java.lang.String
NMTOKENS	java.lang.String[]
ID	java.lang.String
IDREF	java.lang.String
ENTITY	java.lang.String
IDREFS	java.lang.String[]
ENTITIES	java.lang.String[]
nonPositiveInteger	java.math.BigInteger
nonNegativeInteger	java.math.BigInteger
negativeInteger	java.math.BigInteger
unsignedLong	java.math.BigInteger
positiveInteger	java.math.BigInteger
unsignedInt	long
unsignedShort	int
unsignedByte	short
Qname	javax.xml.namespace.QName

Java-to-XML Mapping for Built-In Data Types

For a list of the supported non-built-in Java data types, see "Supported Java Non-Built-In Data Types" on page 6-19.

Table 5-4 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema Data Type
int	int
short	short
long	long
float	float
double	double
byte	byte
boolean	boolean
char	string (with facet of length=1)
java.lang.Integer	int
java.lang.Short	short
java.lang.Long	long
java.lang.Float	float
java.lang.Double	double
java.lang.Byte	byte
java.lang.Boolean	boolean
java.lang.Character	string (with facet of length=1)
java.lang.String	string
java.math.BigInteger	integer
java.math.BigDecimal	decimal
java.lang.String	string
java.util.Calendar	dateTime

Table 5-4 Mapping Java Data Types to XML Schema Data Types

Java Data Type (lower case indicates a primitive data type)	Equivalent XML Schema Data Type
java.util.Date	dateTime
byte[]	base64Binary
weblogic.xml.schema.binding.util.Duration	duration
javax.xml.namespace.QName	Qname

CHAPTER 6

Assembling WebLogic Web Services Using Ant Tasks

The following sections describe how to assemble and deploy WebLogic Web Services using a variety of Ant tasks:

- "Overview of Assembling WebLogic Web Services Using Ant Tasks" on page 6-1
- "Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3
- "Assembling WebLogic Web Services Using Individual Ant Tasks" on page 6-5
- "The Web Service EAR File Package" on page 6-16
- "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16
- "Non-Roundtripping of Generated Data Type Components" on page 6-20
- "Deploying and Testing WebLogic Web Services" on page 6-21

Overview of Assembling WebLogic Web Services Using Ant Tasks

Assembling a WebLogic Web Service refers to gathering all the components of the service (such as the EJB JAR file, the SOAP message handler classes, and so on), generating the `web-services.xml` deployment descriptor file, and packaging everything into an Enterprise Application Archive (EAR) file that can be deployed on WebLogic Server.

There are two ways to assemble a WebLogic Web Service using Ant tasks:

- Using the `servicegen` Ant task, which performs all assembly steps for you.

The `servicegen` Ant takes as input an EJB JAR file (for EJB-implemented Web Services) or a list of Java classes (for Java class-implemented Web Services), and based on information after introspecting the Java code and the attributes of the Ant task, it automatically generates all the components that make up a WebLogic Web Service and packages them into an EAR file.

For detailed information, see "Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3.

- Using a variety of narrowly-focused Ant tasks, such as `autotype`, `source2wsdd`, and so on.

  Typically, the `servicegen` Ant task is adequate for assembling most WebLogic Web Services. If, however, you want more control over how your Web Service is assembled, you can use a set of narrowly-focused Ant tasks instead. For example, you can use the `source2wsdd` to generate the `web-services.xml` file, and then you can update this file manually if you want to add more information.

  For detailed information, see "Assembling WebLogic Web Services Using Individual Ant Tasks" on page 6-5.

For detailed reference information on the Web Services Ant tasks, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or `ant.bat` (Windows) configuration files in the *`WL_HOME`*`\server\bin` directory to set various Ant-specific variables, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation If you need to update these Ant variables, make the relevant changes to the appropriate file for your operating system.

Examples of Assembling WebLogic Web Services

WebLogic Server includes examples of assembling WebLogic Web Services in the *`WL_HOME`*`/samples/server/examples/src/examples/webservices` directory, where *`WL_HOME`* refers to the main WebLogic Platform directory. For detailed instructions on how to build and run the examples, open the following Web page in your browser:

`WL_HOME``/samples/server/examples/src/examples/webservices/package-summary.html`

Assembling WebLogic Web Services Using the servicegen Ant Task

The `servicegen` Ant task takes as input an EJB JAR file or list of Java classes, creates all the needed Web Service components, and packages them into a deployable EAR file.

What the servicegen Ant Task Does

In particular, the `servicegen` Ant task:

- Introspects the Java code, looking for public methods to convert into Web Service operations and non-built-in data types used as parameters or return values of the methods.
- Creates a `web-services.xml` deployment descriptor file, based on the attributes of the `servicegen` Ant task and introspected EJB or Java class information.
- Optionally creates the serialization class that convert the non-built-in data between its XML and Java representations. It also creates XML Schema representations of the Java objects and updates the `web-services.xml` file accordingly. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.
- Packages all the Web Service components into a Web application WAR file, then packages the WAR and EJB JAR files into a deployable EAR file.

Assembling WebLogic Web Services Automatically: Main Steps

To assemble a Web Service automatically using the `servicegen` Ant task:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a staging directory to hold the components of your Web Service.

3. If the Web Service operations are implemented with EJBs, package them, along with any supporting EJBs, into an EJB JAR file. If the operations are implemented with Java classes, compile them into class files.

   For detailed information, refer to *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/programming/environment.html.

4. Copy the EJB JAR file and/or Java class files to the staging directory.

5. In the staging directory, create the Ant build file (called `build.xml` by default) that contains a call to the `servicegen` Ant task.

   For details about specifying the `servicegen` Ant task, see "Creating the Build File That Specifies the servicegen Ant Task" on page 6-4.

   For general information about creating Ant build files, see http://jakarta.apache.org/ant/manual/.

6. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument (if you have created the `build.xml` file to take arguments):

   ```
   prompt> ant
   ```

   The Ant task generates the Web Services EAR file in the staging directory which you can then deploy on WebLogic Server.

Creating the Build File That Specifies the servicegen Ant Task

The following sample `build.xml`, file taken from the `examples.webservices.basic.statelessession` product example, specifies that you will run the `servicegen` Ant task:

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="ws_basic_statelessSession.ear"
       contextURI="WebServices" >
       <service
         ejbJar="HelloWorldEJB.jar"
         targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
         serviceName="HelloWorldEJB"
         serviceURI="/HelloWorldEJB"
         generateTypes="True"
         expandMethods="True"
         style="rpc" >
       </service>
```

```
    </servicegen>
  </target>
</project>
```

In the example, the `servicegen` Ant task creates one Web Service called `HelloWorldEJB`. The URI to identify this Web Service is `/HelloWorldEJB`; the full URL to access the Web Service is

```
http://host:port/WebServices/HelloWorldEJB
```

The `servicegen` Ant task packages the Web Service in an EAR file called `ws_basic_statelessSession.ear`, as specified by the `destEar` attribute. The EAR file contains a WAR file called `web-services.war` (default name) that contains all the Web Service components, such as the `web-services.xml` deployment descriptor file.

Because the `generateTypes` attribute is set to `True`, the WAR file also contains the serialization class for any non-built-in data types used as parameters or return values to the EJB methods. The Ant task introspects the EJBs contained in the `HelloWorldEJB.jar` file, looking for public operations and non-built-in data types, and updates the `web-services.xml` operation and data type mapping sections accordingly. Because the `expandMethods` attribute is also set to `True`, the Ant task lists each public EJB method as a separate operation in the `web-services.xml` file.

The `style="rpc"` attribute specifies that the operations in the Web Service are all RPC-oriented. If the operations in your Web Service are document-oriented, specify `style="document"`.

Note: BEA recommends that you create an exploded directory, rather than an EAR file, by specifying a value for the `destEar` attribute of `servicegen` that does *not* have an `.ear` suffix. You can later package the exploded directory into an EAR file when you are ready to deploy the Web Service.

Assembling WebLogic Web Services Using Individual Ant Tasks

Typically, the `servicegen` Ant task is adequate for assembling most WebLogic Web Services. If, however, you want more control over how your Web Service is assembled, you can use a set of narrowly-focused Ant tasks instead. For example, you can use the `source2wsdd` to generate the `web-services.xml` file, and then you can update this file manually if you want to add more information.

The following sections describe two ways to assemble a Web Service, based on whether you started with Java or with an XML Schema:

- "Assembling a Web Service Starting with Java" on page 6-6
- "Assembling a Web Service Starting with an XML Schema" on page 6-6

Assembling a Web Service Starting with Java

In the following procedure, it is assumed that you have already implemented your Web Service by writing the Java code of the back-end components and non-built-in data types, and you want to use individual Ant tasks to generate the XML Schema that represents the non-built-in data types, as well as the other Web Service components such as the `web-services.xml` deployment descriptor file.

1. Package or compile the Java back-end components that implement the Web Service into their respective packages. For example, package stateless session EJBs into an EJB JAR file and Java classes into class files.

   For detailed instructions, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/programming/environment.html.

2. Create the Web Service deployment descriptor file (`web-services.xml`).

   If you implemented your Web Service with a stateless session EJB or a Java class, you can use the `source2wsdd` Ant task to generate a `web-services.xml` file. For details, see "Running the source2wsdd Ant Task" on page 6-8.

   If you used the `wsdl2Service` Ant task to generate a partial implementation of a Web Service from an existing WSDL file, then the Ant task already generated a `web-services.xml` file for you. For details, see "Generating a Partial Implementation From a WSDL File" on page 5-6.

3. If your Web Service uses non-built-in data types, create all the needed components, such as the serialization class and the XML Schema, by using the `autotype` Ant task to generate these components automatically, as described in "Running the autotype Ant Task" on page 6-9.

4. Optionally create a client JAR file using the `clientgen` Ant task.

   See "Running the clientgen Ant Task" on page 6-12.

5. Package all components into a deployable EAR file by using the `wspackage` Ant task, as described in "Running the wspackage Ant task" on page 6-14.

Assembling a Web Service Starting with an XML Schema

In the following procedure, it is assumed that you are starting with an XML Schema that describes the non-built-in data types of your Web Service, and you want to use the individual Ant tasks to generate the equivalent Java representation of the data types. You then use these generated Java classes to write the back-end component that implements your Web Service, then

use Ant tasks to generate the remaining components, such as the `web-services.xml` file. It is assumed that you want to preserve the original XML Schema all the way through the development process, so that at the end, when you deploy the Web Service, the published WSDL contains the exact same XML Schema with which you stared.

1. Run the `autotype` Ant task to generate the Java representations of the non-built-in data types in the XML Schema file. The Ant task also generates the serialization class used to convert the data between XML and Java and the data type mapping file. Use the `schemaFile` attribute of the `autotype` Ant task to specify the name of the file that contains your XML Schema.

   For details, see "Running the autotype Ant Task" on page 6-9.

2. Write the Java code for the stateless session EJB or Java class back-end component that implements your Web Service. Use the Java classes generated by the `autotype` Ant task in the preceding step for the non-built-in data types (originally described in the XML Schema file from which you started) used as parameters or return values of the methods.

   For details, see "Writing the Java Code for the Components" on page 5-3.

3. If necessary, re-run the `autotype` Ant task against your EJB or Java class to generate the non-built-in data type components for any *new* data types you might have created that are not included in the original XML Schema file. Use the `javaComponents` attribute of the `autotype` Ant task to specify the back-end component you wrote in Step 2.

   Be sure you also use the `typeMappingFile` attribute to specify the existing data type mapping file, generated from the first execution of the `autotype` Ant task in Step 1. The `autotype` Ant task merges the existing XML Schema with any generated one, thus preserving the original XML Schema.

   For details, see "Running the autotype Ant Task" on page 6-9.

4. Run the `source2wsdd` Ant task to generate the `web-services.xml` deployment descriptor. If you re-ran the autotype Ant task to create a merged data type mapping file, be sure you specify this final file with the `typesInfo` attribute.

   For details, see "Running the source2wsdd Ant Task" on page 6-8.

5. Optionally create a client JAR file by running the `clientgen` Ant task.

   For details, see "Running the clientgen Ant Task" on page 6-12.

6. Package all components into a deployable EAR file by using the `wspackage` Ant task.

   For details, see "Running the wspackage Ant task" on page 6-14.

Running the source2wsdd Ant Task

Use the `source2wsdd` Ant task to generate a `web-services.xml` deployment descriptor file from the stateless session EJB or Java source file that implements a Web Service.

To run the `source2wsdd` Ant task:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the `source2wsdd` Ant task. For details, see the examples later in this section.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `source2wsdd` Ant task, see "source2wsdd" on page B-41.

The following example shows a simple `build.xml` file.

Listing 6-1 Simple Source2wsdd build.xml File For a Java Source File

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <source2wsdd
           javaSource="source/MyService.java"
           typesInfo="autotype/types.xml"
           ddFile="ddfiles/web-services.xml"
           serviceURI="/MyService" />
</project>
```

When you run the `source2wsdd` Ant task using the preceding `build.xml` file, the Ant task generates a `web-services.xml` file from the Java source file `source/MyService.java`. It uses non-built-in data type information from the `autotype/types.xml` file; this information includes the XML Schema representation of non-built-in data types used as parameters or return values in your Web Service, as well as data type mapping information that specifies the location of the serialization class, and so on. You typically generate the `types.xml` file using the `autotype` Ant task.

The `source2wsdd` Ant task outputs the generated deployment descriptor information into the file `ddfiles/web-services.xml`. The URI of the Web Service is `/MyService`, used in the full URL that invokes the Web Service once it is deployed.

The following example shows how to generate both a `web-services.xml` file *and* the WSDL file (called `wsdFiles/Temperature.wsdl`) that describes a stateless session EJB-implemented Web Service. Because the `ejbLink` attribute is specified, the `javaSource` attribute must point to the EJB source file. The `source2wsdd` Ant task uses the value of the `ejbLink` attribute as the value of the `<ejb-link>` child element of the `<stateless-ejb>` element in the generated `web-services.xml` file.

Listing 6-2 Source2wsdd build.xml File for an EJB

```
<source2wsdd
       javaSource="source/TemperatureService.java"
       ejbLink="TemperatureService.jar#TemperatureServiceEJB"
       ddFile="ddfiles/web-services.xml"
       typesInfo="autotype/types.xml"
       serviceURI="/TemperatureService"
       wsdlFile="wsdlFiles/Temperature.wsdl"
/>
```

Running the autotype Ant Task

Use the `autotype` Ant task to generate non-built-in data type components, such as the serialization class. For the list of supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.

To run the `autotype` Ant task:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the `autotype` Ant task. For details, see the examples later in this section.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `autotype` Ant task, see "autotype" on page B-7.

The following example shows a simple `build.xml` file.

Listing 6-3 Autotype build.xml File For a Java Class

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
     <autotype  javatypes="mypackage.MyType"
                targetNamespace="http://www.foobar.com/autotyper"
                packageName="a.package.name"
                destDir="output" />
   </target>
</project>
```

When you run the `autotype` Ant task using the preceding `build.xml` file, the Ant task creates the non-built-in data type components for a Java class called `mypackage.MyType`. The package name used in the generated serialization class is `a.package.name`. The serialization Java class and XML schema information is generated and placed in the `output` directory. The generated

XML Schema and type-mapping information are in a file called `types.xml` in this output directory.

The following excerpt from a sample `build.xml` file shows another way to use the `autotype` task

Listing 6-4 Autotype build.xml File For Starting with WSDL

```
<autotype  wsdl="file:/wsdls/myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="output" />
```

The preceding example is similar to the first, except that instead of starting with a Java representation of a data type, the example starts with an XML Schema representation embedded within the WSDL of a Web Service. In this case, the task generates the corresponding Java representation.

Similarly, if you want to start from an XML Schema file and generate the corresponding Java components, use the `schemaFile` attribute, as shown in the following example.

Listing 6-5 Autotype build.xml File for Starting with XML Schema

```
<autotype  schemaFile="file:/schemas/mySchema.xsd"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="output" />
```

In the preceding example, the XML Schema in the `mySchema.xsd` file is copied, without any changes, to the `output/types.xml` file that contains the data type mapping information.

The following example shows how to both generate non-built-in data type components for newly implemented Java data types, and carry forward already generated components.

Listing 6-6 Autotype build.xml File That Carries Forward Existing Components

```
<autotype  javaComponents="my.superService"
           typeMappingFile="file:/mapfiles/types.xml"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="output" />
```

In the preceding example, it is assumed that you have previously run the `autotype` Ant task against an XML Schema file and generated the corresponding Java data types and data type mapping file. It is further assumed that you then used these data types to implement a stateless session EJB back-end component, and during that process, you created additional Java data types and want to use the `autotype` Ant task to generate the corresponding XML Schema. However, you want to preserve the original XML Schema from which you started, and carry it forward into the newly generated `types.xml` file. The example uses the `javaComponents` attribute to specify the EJB which has the new Java data types and the `typeMappingFile` attribute to specify the existing file that contains the XML Schema for data types that you do not want regenerated. The Ant task merges the existing data type mapping information in the `file:/mapfiles/types.xml` file with the generated information, and writes the result to the `output/types.xml` file.

Running the clientgen Ant Task

To run the `clientgen` Ant task and automatically generate a client JAR file:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the `clientgen` Ant task. For details, see the examples later in this section.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `clientgen` Ant task, see "clientgen" on page B-14.

The following example shows a simple `build.xml` file.

Listing 6-7 Clientgen build.xml File For Generating Client From EAR File

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
      <clientgen ear="myapps/myapp.ear"
                 serviceName="myService"
                 packageName="myapp.myservice.client"
                 useServerTypes="True"
                 clientJar="myapps/myService_client.jar" />
   </target>
</project>
```

When you run the `clientgen` Ant task using the preceding `build.xml` file, the Ant task creates the `myapps/myService_client.jar` client JAR file that contains the service-specific client interfaces and stubs and the serialization class used to invoke the WebLogic Web Service called `myService` contained in the EAR file `myapps/myapp.ear`. It packages the client interface and stub files into a package called `myapp.myservice.client`. The `useServerTypes` attribute specifies that the `clientgen` Ant task should get the Java implementation of all non-built-in data types used in the Web Service from the `myapps/myapp.ear` file rather than generating Java code to implement the data types.

The following excerpt from a sample `build.xml` file shows another way to use the `clientgen` task.

Listing 6-8 Clientgen build.xml File For Generating Client From a WSDL File

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
```

```
            clientJar="myapps/myService_client.jar"
/>
```

In the example, the `clientgen` task creates a client JAR file (called `myapps/myService_client.jar`) to invoke the Web Service described in the `http://example.com/myapp/myservice.wsdl` WSDL file. It packages the interface and stub files in the `myapp.myservice.client` package.

Running the wspackage Ant task

Use the `wspackage` Ant task to package the various components of a Web Service into a new deployable EAR file or to add additional components to an existing EAR file.

To run the `wspackage` Ant task, follow these steps:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the `wspackage` Ant task. For details, see the examples later in this section.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

   ```
   prompt> ant
   ```

For reference information about the `wspackage` Ant task, see "wspackage" on page B-52.

The following example shows a simple `build.xml` file for creating a new deployable EAR file for a Java class-implemented Web Service.

Listing 6-9 Wspackage build.xml File for a Java Class

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <wspackage
           output="ears/myWebService.ear"
           contextURI="web_services"
           codecDir="autotype"
           webAppClasses="example.ws2j.service.SimpleTest"
           ddFile="ddfiles/web-services.xml" />
</project>
```

When you run the `wspackage` Ant task using the preceding `build.xml` file, the Ant task creates an EAR file called `ears/myWebService.ear`. The context URI of the Web Service, used in the full URL that invokes it, is `web_services`. The class file that contains the serialization class for the non-built-in data types is located in the `autotype` directory. The Java class that implements the Web Service is called `example.ws2j.service.SimpleTest` and will be packaged in the `WEB-INF/classes` directory of the Web application. Finally, the existing deployment descriptor file is `ddfiles/web-services.xml`.

The following example shows a another `build.xml` file for adding additional components to an existing EAR file.

Listing 6-10 Wspackage build.xml File For Adding Additional Components to EAR File

```
<project name="buildWebservice" default="generate-typeinfo">
   <target name="generate-typeinfot">
      <wspackage
           output="ears/myWebService.ear"
           overwrite="false"
           filesToEar="myEJB2.jar"
       />
</project>
```

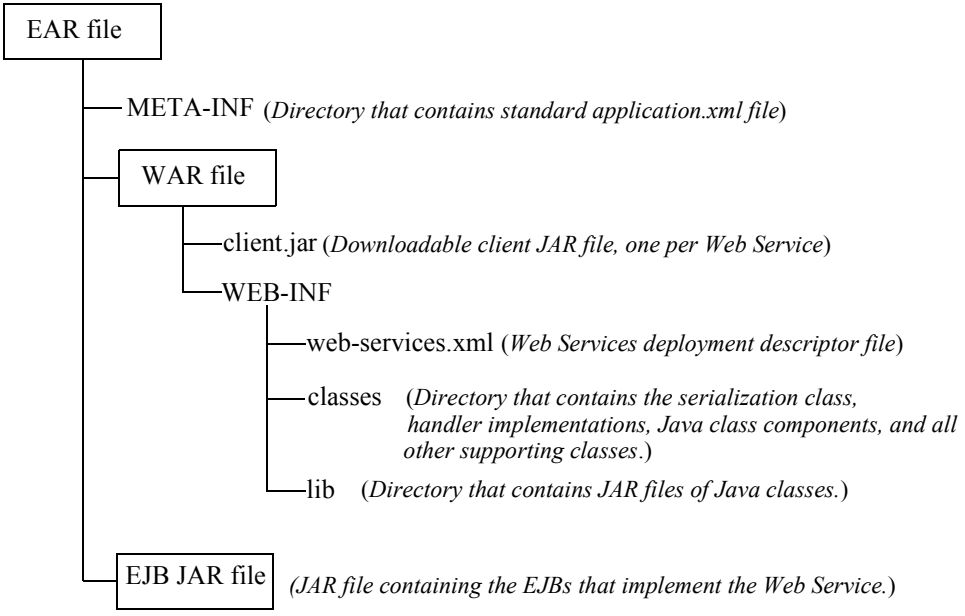
In the example, the `wspackage` Ant task adds the file `myEJB2.jar` to the root directory of the existing EAR file called `ears/myWebService.ear`. The `overwrite` attribute is set to `false` to ensure that none of the existing components in the `ears/myWebService.ear` file are overwritten; be sure you always do this when using the `wspackage` Ant task to add additional files to an EAR file.

The Web Service EAR File Package

Web Services are packaged into standard Enterprise Application EAR files that contain a Web application WAR file along with the EJB JAR files.

The following graphic shows the hierarchy of a typical WebLogic Web Services EAR file.

Figure 6-1 Hierarchy of WebLogic Web Services EAR File


EAR file
META-INF (Directory that contains standard application.xml file)
WAR file
client.jar (Downloadable client JAR file, one per Web Service)
WEB-INF
web-services.xml (Web Services deployment descriptor file)
classes (Directory that contains the serialization class, handler implementations, Java class components, and all other supporting classes.)
lib (Directory that contains JAR files of Java classes.)
EJB JAR file
(JAR file containing the EJBs that implement the Web Service.)

Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks

The tables in the following sections list the non-built-in XML and Java data types for which the `servicegen` and `autotype` Ant tasks can generate data type components, such as the serialization class, the Java or XML representation, and so on.

If your XML or Java data type is not listed in these tables, and it is not one of the built-in data types listed in "Supported Built-In Data Types" on page 5-15, then you must create the non-built-in data type components manually. For details, see Chapter 11, "Using Non-Built-In Data Types."

Warning: The serialization class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

For information on the ways that WebLogic Web Services are non-compliant with the JAX-RPC specification with respect to data types, see "Data Type Non-Compliance with JAX-RPC" on page 6-20.

Supported XML Non-Built-In Data Types

The following table lists the supported XML Schema non-built-in data types. If your XML data type is listed in the table, then the `servicegen` and `autotype` Ant tasks can generate the serialization class to convert the data between its XML and Java representations, as well as the Java representation and type mapping information for the `web-services.xml` deployment descriptor.

For details and examples of the data types, see the JAX-RPC specification.

Table 6-1 Supported Non-Built-In XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
Enumeration	Typesafe enumeration pattern. For details, see Section 4.2.4 of the JAX-RPC specification.
`<xsd:complexType>` with elements of both simple and complex types.	JavaBean
`<xsd:complexType>` with simple content.	JavaBean
`<xsd:attribute>` in `<xsd:complexType>`	Property of a JavaBean

Table 6-1 Supported Non-Built-In XML Schema Data Types

XML Schema Data Type	Equivalent Java Data Type or Mapping Mechanism
Derivation of new simple types by restriction of an existing simple type.	Equivalent Java data type of simple type.
Facets used with restriction element. **Note:** The base primitive type must be one of the following: `string`, `decimal`, `float`, or `double`. Pattern facet is not enforced.	Restriction enforced during serialization and deserialization.
`<xsd:list>`	Array of the list data type.
Array derived from `soapenc:Array` by restriction using the `wsdl:arrayType` attribute.	Array of the Java equivalent of the `arrayType` data type.
Array derived from `soapenc:Array` by restriction.	Array of Java equivalent.
Derivation of a complex type from a simple type.	JavaBean with a property called `simpleContent` of type `String`.
`<xsd:anyType>`	java.lang.Object.
`<xsd:nil>` and `<xsd:nillable>` attribute	Java `null` value. If the XML data type is built-in and usually maps to a Java primitive data type (such as `int` or `short`), then the XML data type is actually mapped to the equivalent object wrapper type (such as `java.lang.Integer` or `java.lang.Short`).
Derivation of complex types by extension	Mapped using Java inheritance.
Abstract types	Abstract Java data type.

Supported Java Non-Built-In Data Types

The following table lists the supported Java non-built-in data types. If your Java data type is listed in the table, then the `servicegen` and `autotype` Ant tasks can generate the serialization class to convert the data between its Java and XML representations.

Table 6-2 Supported Non-Built-In Java Data Types

Java Data Type	Equivalent XML Schema Data Type
Array of any supported data type.	SOAP Array.
JavaBean whose properties are any supported data type.	<xsd:sequence>
java.util.List	SOAP Array.
java.util.ArrayList	SOAP Array.
java.util.LinkedList	SOAP Array.
java.util.Vector	SOAP Array.
java.util.Stack	SOAP Array.
java.util.Collection	SOAP Array.
java.util.Set	SOAP Array.
java.util.HashSet	SOAP Array.
java.util.SortedSet	SOAP Array.
java.util.TreeSet	SOAP Array
java.lang.Object **Note:** The data type of the runtime object must be a known type: either a built-in data type or one that has type mapping information.	<xsd:anyType>
JAX-RPC-style enumeration class	<xsd:simpleType> with enumeration facets

Data Type Non-Compliance with JAX-RPC

The `autotype` Ant task does not comply with the JAX-RPC specification if the XML Schema data type (for which it is generating the Java representation) has all the following characteristics:

- The data type is a `complexType`.
- The `complexType` contains a single `sequence`.
- The `sequence` contains a single `element` with `maxOccurs` greater than 1 or unbounded.

The following example shows such an XML Schema data type:

```
<xsd:complexType name="Response">
    <xsd:sequence >
        <xsd:element name="code" type="xsd:string" maxOccurs="10" />
    </xsd:sequence>
</xsd:complexType>
```

The `autotype` Ant task maps this type of XML Schema data type directly to a Java array of the specified element. In the previous example, the `autotype` Ant task maps the `Response` XML Schema data type to a `java.lang.String[]` Java type. This is similar to the type of mapping that .NET does.

The JAX-RPC specification, in turn, states that this type of XML Schema data type should map to a Java array with a pair of setter and getter methods *in a JavaBean class*. WebLogic Web Services do not follow this last part of the specification.

Non-Roundtripping of Generated Data Type Components

When you use the `servicegen` or `autotype` Ant tasks to create the serialization class and Java or XML representation of non-built-in data types, note that the process cannot be round-tripped. This means that if, for example, you use the `autotype` Ant task to generate the Java representation of an XML Schema data type, and then use `autotype` to create an XML Schema data type from the generated Java type, the original and generated XML Schema data type will not necessarily look the same, although they both describe the same XML data. This is also true if you start from Java, generate an XML Schema, then generate a new Java data type from the generated XML Schema: the original and generated Java type will not necessarily look exactly the same. For example, the original and generated Java type might list the parameters of the constructor in a different order.

This behavior has a variety of repercussions. For example, assume you are developing a Web Service from an existing stateless session EJB that uses non-built-in data types. You use the

`autotype` Ant task to generate the serialization class and Java and XML representation of the data types and you use this generated code in your server-side code that implements your Web Service. Later you use the `clientgen` Ant task to generate the Web Service-specific client JAR file, which also includes a serialization class and the Java representation of the non-built-in data types. However, because `clientgen` by default generates these components from the WSDL of the Web Service (and thus from an XML Schema), the `clientgen`-generated client-side Java representation might look different from the `autotype`-generated server-side Java code. This means that you might not necessarily be able to reuse any server-side code that handles the data type in your client application. If you want the `clientgen` Ant task to always use the generated serialization class and code from the WebLogic Web Service EAR file, specify the `useServerTypes` attribute.

Deploying and Testing WebLogic Web Services

Deploying a WebLogic Web Service refers to making it available to remote clients. Because WebLogic Web Services are packaged as standard J2EE Enterprise applications, deploying a Web Service is the same as deploying an Enterprise application.

For detailed information on deploying Enterprise applications, see *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/deployment/index.html.

You can use the WebLogic Web Services Home Page to test your Web Services, as described in "WebLogic Web Services Home Page and WSDL URLs" on page 6-21.

WebLogic Web Services Home Page and WSDL URLs

Every Web Service deployed on WebLogic Server has a Home Page. From the Home page you can:

- View the WSDL that describes the service.
- Test each operation to ensure that it is working correctly.
- View the SOAP request and response messages from a successful test of an operation.

The following URLs show first how to invoke the Web Service Home page and then the WSDL in your browser:

```
[protocol]://[host]:[port]/[contextURI]/[serviceURI]
[protocol]://[host]:[port]/[contextURI]/[serviceURI]?WSDL
```

where:

- *protocol* refers to the protocol over which the service is invoked, either HTTP or HTTPS. This value corresponds to the `protocol` attribute of the `<web-service>` element that describes the Web Service in the `web-servicex.xml` file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `protocol` attribute.
- *host* refers to the computer on which WebLogic Server is running.
- *port* refers to the port number on which WebLogic Server is listening (default value is `7001`).
- *contextURI* refers to the context root of the Web application, corresponding to the `<context-root>` element in the `application.xml` deployment descriptor of the EAR file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `contextURI` attribute.

  If your `application.xml` file does not include the `<context-root>` element, then the value of `contextURI` is the name of the Web application archive file or exploded directory.
- *serviceURI* refers to the URI of the Web Service. This value corresponds to the `uri` attribute of the `<web-service>` element in the `web-services.xml` file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `serviceURI` attribute.

For example, assume you used the following `build.xml` file to assemble a WebLogic Web Service using the `servicegen` Ant task:

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
      <servicegen
         destEar="myWebService.ear"
         warName="myWAR.war"
         contextURI="web_services">
         <service
             ejbJar="myEJB.jar"
             targetNamespace="http://www.bea.com/examples/Trader"
             serviceName="TraderService"
             serviceURI="/TraderService"
             generateTypes="True"
             expandMethods="True" >
         </service>
     </servicegen>
```

```
  </target>
</project>
```

The URL to invoke the Web Service Home Page, assuming the service is running on a host called `ariel` at the default port number, is:

```
http://ariel:7001/web_services/TraderService
```

The URL to get the automatically generated WSDL of the Web Service is:

```
http://ariel:7001/web_services/TraderService?WSDL
```

Denying Access to the WSDL and Home Page of a WebLogic Web Service

Because the WSDL defines the public contract of a Web Service, you usually want to make it readily available. By default, the WSDL that describes a WebLogic Web Service is always publicly accessible. The WSDL is also accessible from the Web Service's Home Page.

You might, however, sometimes want to turn off public access to the WSDL or the Home Page. To do this, update the appropriate `<web-service>` element of the `web-services.xml` deployment descriptor file that describes the WebLogic Web Service, adding the attribute `exposeWSDL="False"` or `exposeHomePage="False"`, as shown in the following excerpt:

```
<web-service targetNamespace="http://example.com"
             name="myorderproc"
             uri="myOrderProcessingService"
             exposeWSDL="False"
             exposeHomePage="False">
...
</web-service>
```

You must redeploy the Web Service after updating its deployment descriptor for the change to take effect.

CHAPTER 7

Invoking Web Services from Client Applications and WebLogic Server

The following sections describe how to invoke Web Services, both WebLogic and non-WebLogic, from client applications and from WebLogic Server:

- "Overview of Invoking Web Services" on page 7-2
- "Creating Java Client Applications to Invoke Web Services: Main Steps" on page 7-4
- "Writing an Asynchronous Client Application" on page 7-11
- "Using Web Services System Properties" on page 7-14
- "Using a Proxy Server with the WebLogic Web Services Client" on page 7-24
- "Invoking Web Services from WebLogic Server" on page 7-22
- "Writing Advanced Java Client Applications" on page 7-25

For information about invoking a WebLogic Web Service using reliable SOAP messaging, see Chapter 10, "Using Reliable SOAP Messaging."

It is assumed in this chapter that the client applications use HTTP/S as the connection protocol when invoking a WebLogic Web Service. You can, however, configure your Web Service so that client applications can also use JMS as the transport when invoking the Web Service. For details, see Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service."

Overview of Invoking Web Services

Invoking a Web Service refers to the actions that a client application performs to use the Web Service. Client applications that invoke Web Services can be written using any technology: Java, Microsoft SOAP Toolkit, Microsoft .NET, and so on.

Note: This chapter uses the term *client application* to refer to both a standalone client that uses the WebLogic thin client to invoke a Web Service hosted on both WebLogic and non-WebLogic Servers, and a client that runs inside of an EJB running on WebLogic Server.

The sections that follow describe how to use BEA's implementation of the JAX-RPC specification (Version 1.0) to invoke a Web Service from a Java client application. You can use this implementation to invoke Web Services running on any server, both WebLogic and non-WebLogic. In addition, you can create a standalone client application or one that runs as part of a WebLogic Server.

WebLogic Server provides optional runtime client JAR files that include, for your convenience when developing a standalone client application, the classes you need to invoke a Web Service. You can also use the `clientgen` Ant task to generate a Web Service-specific JAR file that contains the stubs, defined by the JAX-RPC specification, that client applications use to statically invoke a Web Service. These stubs implement JAX-RPC interfaces such as `Stub` and `Service`.

For information about troubleshooting problems when invoking a Web Service, see Chapter 20, "Troubleshooting."

JAX-RPC API 1.0

The Java API for XML based RPC (JAX-RPC) is a Sun Microsystems specification that defines the Web Services APIs.

The following table briefly describes the core JAX-RPC interfaces and classes.

Table 7-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Service	Main client interface.
ServiceFactory	Factory class for creating `Service` instances.

Table 7-1 JAX-RPC Interfaces and Classes

javax.xml.rpc Interface or Class	Description
Stub	Base class of the client proxy used to invoke the operations of a Web Service.
Call	Used to dynamically invoke a Web Service.
JAXRPCException	Exception thrown if an error occurs while invoking a Web Service.

WebLogic Server implements the JAX-RPC 1.0 specification.

For detailed information on JAX-RPC, see http://java.sun.com/xml/jaxrpc/index.html.

For a tutorial that describes how to use JAX-RPC to invoke Web Services, see http://java.sun.com/webservices/docs/ea1/tutorial/doc/JAXRPC.html.

The Runtime Client JAR Files

WebLogic Server provides the following runtime client JAR files for use with standalone client applications (that is, client applications that do not run in a WebLogic Server instance). These JAR files are located in the *WL_HOME*`/server/lib` directory, where *WL_HOME* refers to the top-level directory of WebLogic Platform.

- `webserviceclient.jar`: Contains the client runtime implementation of JAX-RPC.
- `webserviceclient+ssl.jar`: Same as `webserviceclient.jar`, plus the runtime implementation of SSL.

  Use this runtime client JAR file if you are using SSL to secure your Web Service and you want to use the WebLogic Server-provided implementation of the SSL client classes.
- `webserviceclient+ssl_pj.jar`: Same as the `webserviceclient_ssl.jar`, but for the CDC profile of J2ME.

  Use this runtime client JAR file if you are writing a J2ME client that uses SSL.

Client applications that use the `webserviceclient.jar` file (or the SSL and J2ME variants) should *not* have `webservices.jar` or `weblogic.jar` in their CLASSPATH. All classes needed to run a client application that invokes a Web Service are typically available in `webserviceclient.jar`. If, however, there are some other classes needed by your application

that are missing from `webserviceclient.jar`, but are included in `webservices.jar` or `weblogic.jar`, then put these JAR files *after* `webserviceclient.jar` in your CLASSPATH.

Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

Examples of Clients That Invoke Web Services

WebLogic Server includes the following examples of creating and invoking WebLogic Web Services in the *`WL_HOME`*`/samples/server/src/examples/webservices` directory, where *`WL_HOME`* refers to the main WebLogic Platform directory:

- `basic.statelessSession`: Uses a stateless session EJB back-end component with built-in data types as its parameters and return value.
- `basic.javaclass`: Uses a Java class back-end component with built-in data types as its parameters and return value.
- `complex.statelessSession`: Uses a stateless session EJB back-end component with non-built-in data types as its parameters and return value.
- `handler.log`: Uses both a handler chain and a stateless session EJB.
- `handler.nocomponent`: Uses only a handler chain with *no* back-end component.
- `client.static`: Shows how to create a static client application that invokes a non-WebLogic Web Service.
- `client.dynamic_wsdl`: Shows how to create a dynamic client application that uses WSDL to invoke a non-WebLogic Web Service.
- `client.dynamic_no_wsdl`: Shows how to create a dynamic client application that does not use WSDL to invoke a non-WebLogic Web Service.

For detailed instructions on how to build and run the examples, open the following Web page in your browser:

`WL_HOME``/samples/server/src/examples/webservices/package-summary.html`

Creating Java Client Applications to Invoke Web Services: Main Steps

To create a Java client application that invokes a Web Service, follow these steps:

1. Generate the Web Service-specific client JAR file by running the `clientgen` Ant task.

   Specify the `wsdl` attribute to create a client JAR file for a Web Service that is being hosted by either a WebLogic or a non-WebLogic server, or specify the `ear` attribute for WebLogic Web Services packaged in EAR files.

   For details and examples of running the `clientgen` Ant task, see "Generating the Client JAR File by Running the clientgen Ant Task" on page 7-5. For reference information, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

2. Get information about the Web Service, such as its name and signature.

   For details, see "Getting Information About a Web Service" on page 7-7.

3. Write the Java client application code that includes the invoke of the Web Service.

   See "Writing the Java Client Application to Invoke a Web Service" on page 7-8 for an example of writing a simple client application.

4. Compile and run your Java client application.

   If you are creating a standalone client application, ensure that the `webserviceclient.jar` runtime Java client JAR file provided by WebLogic Server is in your CLASSPATH. For details, see "The Runtime Client JAR Files" on page 7-3.

   If your client application is running on WebLogic Server, you do not need this runtime client JAR file.

Generating the Client JAR File by Running the clientgen Ant Task

The Web Service-specific JAR file contains the stubs, such as implementation of the `Stub` and `Service` interfaces, which are defined by the JAX-RPC specification and are used by client applications to invoke a Web Service (either WebLogic or non-WebLogic). Almost *all* the code you need is automatically generated for you.

Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

To run the `clientgen` Ant task and automatically generate the Web Service-specific client JAR file:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is

BEA_HOME\user_projects\domains\domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

On UNIX, execute the setEnv.sh command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*/user_projects/domains/domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Create a file called build.xml that contains a call to the clientgen Ant task. For details, see the example later in this section.
3. Execute the Ant task or tasks specified in the build.xml file by typing ant in the same directory as the build.xml file:

   ```
   prompt> ant
   ```

For reference information about the clientgen Ant task, see "clientgen" on page B-14.

The following example shows a simple build.xml file.

Listing 7-1 Sample build.xml File for the clientgen Ant Task

```
<project name="buildWebservice" default="generate-client">
   <target name="generate-client">
     <clientgen wsdl="http://example.com/myapp/myservice?WSDL"
                packageName="myapp.myservice.client"
                clientJar="myapps/myService_client.jar"
/>
   </target>
</project>
```

When you run the clientgen Ant task using the preceding build.xml file, the Ant task creates a client JAR file (called myapps/myService_client.jar) that the client application will use to invoke the Web Service described by the http://example.com/myapp/myservice?WSDL WSDL. It packages the interface and stub files in the myapp.myservice.client package.

Getting Information About a Web Service

You need to know the name of the Web Service and the signature of its operations before you write your client code. There are a variety of ways to find this information.

If you are invoking a WebLogic Web Service, you can use its Home Page to get the full signature of each operation. For details, see "Using the Web Service Home Page to Test Your Web Service" on page 20-2.

Another way to get the signature of a Web Service operation is to use the `clientgen` Ant task to generate the Web Service-specific client JAR file, un-JAR the file, and look at the generated `*.java` files. Typically, the file *`ServiceName`*`Port.java` contains the interface definition of your Web Service, where *`ServiceName`* refers to the name of the Web Service. For example, look at the `TraderServicePort.java` file for the signature of the `buy` and `sell` operations.

Finally, you can examine the actual WSDL of the Web Service. The name of the Web Service is contained in the `<service>` element, as shown in the following excerpt of the `TraderService` WSDL:

```
<service name="TraderService">
  <port name="TraderServicePort"
       binding="tns:TraderServiceSoapBinding">
...
  </port>
</service>
```

The operations defined for this Web Service are listed under the corresponding `<binding>` element. For example, the following WSDL excerpt shows that the `TraderService` Web Service has two operations, `buy` and `sell` (for clarity, only relevant parts of the WSDL are shown):

```
<binding name="TraderServiceSoapBinding" ...>
  ...
  <operation name="sell">
  ...
  </operation>
  <operation name="buy">
  </operation>
</binding>
```

Writing the Java Client Application to Invoke a Web Service

The following sections describe how to write Java client applications to invoke a Web Service. The example uses the JAX-RPC API and assumes that you have the necessary BEA-provided JAR files in your CLASSPATH.

Writing a Simple Client Application

You use a strongly-typed Java interface when you use a static client application to invoke a Web Service. The Web Services-specific JAR file includes the following classes and interfaces:

- Implementation of the JAX-RPC `Service` interface for the Web Service you are invoking.
- An implementation of the `Stub` interface for each SOAP port in the WSDL.
- Serialization class for non-built-in data types and their Java representations.

The following code shows an example of writing a client application that invokes the sample `TraderService` Web Service; in the example, `TraderService` is the stub factory and `TraderServicePort` is the stub itself:

```
package examples.webservices.complex.statelessSession;

public class Client {

  public static void main(String[] args) throws Exception {

    // Setup the global JAXM message factory
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // Parse the argument list
    Client client = new Client();
    String wsdl = (args.length > 0? args[0] : null);
    client.example(wsdl);
  }

  public void example(String wsdlURI) throws Exception {

    TraderServicePort trader = null;
    if (wsdlURI == null) {
      trader = new TraderService_Impl().getTraderServicePort();
    } else {
      trader = new TraderService_Impl(wsdlURI).getTraderServicePort();
```

```
    }
    String [] stocks = {"BEAS", "MSFT", "AMZN", "HWP" };

      // execute some buys
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Buying "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.buy(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());
    }
    // execute some sells
    for (int i=0; i<stocks.length; i++) {
      int shares = (i+1) * 100;
      log("Selling "+shares+" shares of "+stocks[i]+".");
      TradeResult result = trader.sell(stocks[i], shares);
      log("Result traded "+result.getNumberTraded()
          +" shares of "+result.getStockSymbol());

    }

  }

  private static void log(String s) {
    System.out.println(s);
  }

}
```

In the preceding example:

- The following code shows how to create a `TraderServicePort` stub:

  ```
  trader = new TraderService_Impl(wsdlURI).getTraderServicePort();
  ```

  The `TraderService_Impl` stub factory implements the JAX-RPC `Service` interface. The constructor of `TraderService_Impl` creates a stub based on the provided WSDL URI. The `getTraderServicePort()` method is used to return an instance of the `TraderService` stub implementation.

- The following code shows how to invoke the `buy` operation of the `TraderService` Web Service:

  ```
  TradeResult result = trader.buy(stocks[i], shares);
  ```

  The `trader` Web Service has two operations: `buy()` and `sell()`. Both operations return a non-built-in data type called `TradeResult`.

Writing a Client That Uses Out or In-Out Parameters

Web Services can use out or in-out parameters as a way of returning multiple values. For more information on out and in-out parameters, see "Implementing Multiple Return Values" on page 5-10.

When you write a client application that invokes a Web Service that uses out or in-out parameters, the data type of the out or in-out parameter must implement the `javax.xml.rpc.holders.Holder` interface. After the client application invokes the Web Service, the client can query the out or in-out parameters in the `Holder` object and treat them as if they were standard return values.

For example, the Web Service described by the following WSDL has an operation called `echoStructAsSimpleTypes()` that takes one standard input parameter and three out parameters:

`http://soap.4s4c.com/ilab/soap.asp?WSDL`

The following client application shows one way to invoke the `echoStructAsSimpleTypes()` Web Service operation.

```
package websvc;

public class Main {

  public static void main(String[] args) throws Exception {
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    InteropLab_Impl test = new InteropLab_Impl();
    InteropTest2PortType soap = test.getinteropTest2PortType();

    org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct inputStruct =
      new org.tempuri.x4s4c.x1.x3.wsdl.types.SOAPStruct();

    inputStruct.setVarInt(10);
    inputStruct.setVarFloat(10.1f);
    inputStruct.setVarString("hi there");

    javax.xml.rpc.holders.StringHolder outputString =
      new javax.xml.rpc.holders.StringHolder();
    javax.xml.rpc.holders.IntHolder outputInteger =
      new javax.xml.rpc.holders.IntHolder();
    javax.xml.rpc.holders.FloatHolder outputFloat =
      new javax.xml.rpc.holders.FloatHolder();
```

```
    soap.echoStructAsSimpleTypes(inputStruct, outputString, outputInteger,
                                 outputFloat);

    System.out.println("This example shows how to create a static client
                        application that invokes a non-WebLogic Web Service.");
    System.out.println("The webservice used was:
                       http://soap.4s4c.com/ilab/soap.asp?WSDL");
    System.out.println("This webservice shows how to invoke an operation that
                       uses out parameters.  The set parameters are below:");
    System.out.println("outputString.value: " + outputString.value);
    System.out.println("outputInteger.value: " + outputInteger.value);
    System.out.println("outputFloat.value: " + outputFloat.value);
  }
}
```

Writing an Asynchronous Client Application

This section describes how to invoke an operation asynchronously. In this context, *asynchronously* means you invoke an operation and then optionally get the results of the invoke in a later step.

Warning: This section applies only to static client application that use the Web Service-specific client JAR files generated by the `clientgen` Ant task. You cannot use the procedure specified in this section in dynamic proxy and DII-based client applications.

To write an asynchronous client, follow these steps:

1. Generate the Web Service-specific client JAR file by running the `clientgen` Ant task. Be sure to specify the `generateAsyncMethods="True"` attribute, as shown in the following example:

   ```
   <clientgen
      wsdl="http://www.mssoapinterop.org/asmx/simple.asmx?WSDL"
      clientJar="echoservice.jar"
      packageName="examples.async"
      generateAsyncMethods="true" />
   ```

   The `clientgen` Ant task generates special asynchronous methods in the JAX-RPC stubs to invoke the operations of the Web Service. See "Description of the Generated Asynchronous Web Service Client Stub" on page 7-12 for more details.

   For general details and examples of running the `clientgen` Ant task, see "Generating the Client JAR File by Running the clientgen Ant Task" on page 7-5. For reference information, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

2. Write the Java code using the special asynchronous methods. For examples, see "Writing the Asynchronous Client Java Code" on page 7-13.
3. Compile and run your asynchronous Java client application.

   If you are creating a standalone asynchronous client application, ensure that the `webserviceclient.jar` runtime Java client JAR file provided by WebLogic Server is in your CLASSPATH. If your client application is running on WebLogic Server (for example, as part of the reliable SOAP messaging framework), you can omit this step.

   For detail about the `webserviceclient.jar` file, as well as the other available runtime client JAR files, see "The Runtime Client JAR Files" on page 7-3.

For detailed API reference information about writing asynchronous client applications, see the weblogic.webservice.async Javadoc.

Description of the Generated Asynchronous Web Service Client Stub

When you specify `generateAsyncMethods="True"` when executing the `clientgen` Ant task, the task creates two special methods in the generated JAX-RPC stub to invoke each Web Service operation asynchronously, in addition to the standard methods. The special methods take the following form:

```
FutureResult startMethod (params, AsyncInfo asyncInfo);
result endMethod (FutureResult futureResult);
```

where:

- *`Method`* is the name of the synchronous method used to invoke the Web Service operation.
- *`params`* is the list of parameters of the operation.
- *`result`* is the result of the operation.
- `FutureResult` is a WebLogic object used as a placeholder for the impending result.
- `AsyncInfo` is a WebLogic object used to pass additional information to WebLogic Server.

Note: If the operations of the Web Service are document-oriented (rather than RPC-oriented), the `clientgen` Ant task also generates the following `end()` method, in addition to the methods listed above:

```
result endConvenienceMethod (FutureResult futureResult);
```

If you use convenience methods when invoking document-oriented Web Service operations, then use this flavor of the `end()` method when invoking the operation asynchronously.

For example, assume the standard generated stub contains the following method to invoke a Web Service operation called `echoString`:

```
String echoString (String str);
```

The `clientgen` task generates the following additional asynchronous methods in the generated stub:

```
FutureResult startEchoString (String str, AsyncInfo asyncInfo);
String endEchoString (FutureResult futureResult);
```

For detailed API reference information about the `FutureResult` interface and the `AsyncInfo` class, see the weblogic.webservice.async Javadoc.

Writing the Asynchronous Client Java Code

When you write a Java client application that asynchronously invokes a Web Service operation, you must first import the following classes:

```
import weblogic.webservice.async.FutureResult;
import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.ResultListener;
import weblogic.webservice.async.InvokeCompletedEvent;
```

There are two steps involved in invoking an asynchronous operation: the first starts the invocation and the second optionally retrieves the results of the completed operation.

Assume that your client application uses the following Java code to get an instance of the `SimpleTest` stub implementation:

```
SimpleTest echoService = new SimpleTest_Impl();
SimpleTestSoap echoPort = echoService.getSimpleTestSoap();
```

Further assume that you want to invoke the `echoString` operation of the Web Service. The following paragraphs show a variety of ways you can invoke this operation asynchronously.

The simplest way is to simply execute the `startEchoString()` client method, do some other task, then execute the `endEchoString()` client method:

```
FutureResult futureResult = echoPort.startEchoString( "94501", null );
// do something
String result = echoPort.endEchoString( futureResult );
```

The end*Method*() method, in this case endEchoString(), blocks until the result is ready.

You can also use the FutureResult.isCompleted() method to test whether the results have returned from the Web Service, as shown in the following excerpt:

```
FutureResult futureResult = echoPort.startEchoString( "94501", null );

while( !futureResult.isCompleted() ){
  // do something ;
}

String result = echoPort.endEchoString( futureResult );
```

Alternatively, you can use the ResultListener and InvokeCompletedEvent classes to set up a listener in your client application that listens for a callback indicating that the results of the operation have returned, as shown in the following excerpt:

```
AsyncInfo asyncInfo = new AsyncInfo();

asyncInfo.setResultListener( new ResultListener(){
  public void onCompletion( InvokeCompletedEvent event ){

    SimpleTestSoap source = (SimpleTestSoap)event.getSource();

    try{
      String result = source.endEchoString ( event.getFutureResult() );
     } catch ( RemoteException e ){
      e.printStackTrace ( System.out );
    }
  }
});

echoPort.startEchoString( "94501", asyncInfo );
```

Using Web Services System Properties

The following two tables list the WebLogic and standard JDK 1.4 system properties you can set in client applications that invoke Web Services. Use the System.setProperty() method to set the properties.

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.transport.http.full-url	Specifies that the full URL, rather than the relative URL, of the Web Service that the client application is invoking be specified in the `Request-URI` field of HTTP request. Valid values are `True` and `False`. Default value is `False`.	Boolean.
weblogic.webservice.transport.https.proxy.host	If you use a proxy server to make HTTPS (HTTP over SSL) connections, use this system property to specify the host name of the proxy server in your client applications.	String.
weblogic.webservice.transport.https.proxy.port	If you use a proxy server to make HTTPS (HTTP over SSL) connections, use this system property to specify the port of the proxy server in your client applications.	String.
weblogic.webservice.verbose	Enables verbose mode during Web Service invocation The SOAP request and response messages are printed to the standard out of the client. Valid values are `True` and `False`. Default value is `False`. For details, see "Viewing SOAP Messages" on page 20-4.	Boolean
weblogic.webservice.client.ssl.strictcertchecking	Enables or disables strict certificate validation when using the WebLogic-provided implementation of SSL. Set to `True` to enable strict certificate validation, and `False` to disable. Default value is `False`. For an example, see "Using the WebLogic Server-Provided SSL Implementation" on page 13-33.	Boolean

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.client.ssl.trustedcertfile	The name of the file (located on the client application computer) that contains the certificates of CA (certificate authority). The CAs are trusted to issue WebLogic Server certificates. The file can also contain certificates that you trust directly.	String
weblogic.webservice.client.ssl.adapterclass	Fully qualified name of an adapter class you have implemented to use a third-party SSL implementation. For an example, see "Using a Third-Party SSL Implementation" on page 13-38.	String.

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.security.clock.precision	Describes the accuracy of synchronization between the clock of the client application invoking a WebLogic Web Service and WebLogic Server's clock. The client application uses this value to account for a reasonable level of clock skew between two clocks. The value is expressed in milliseconds. This means, for example, that if the clocks are accurate within a one minute of each other, the value of this element is 60000. If the value of this element is greater than the expiration period of the SOAP response, the client application rejects the request because it cannot accurately enforce the expiration. For example, if the clock precision value is 60000 milliseconds, and the client application receives a SOAP response that expires 30000 milliseconds after its creation time, it is possible that the message has lived for longer than 30000 seconds, due to the 60000 millisecond clock precision discrepancy, so the client application has no option but to reject the message. You can relax this strict enforcement by setting the `weblogic.webservice.security.clock.precision.lax` property to `false`. This property must be specified in conjunction with `weblogic.webservice.security.clock.synchronized`. The default value for this property is 60000.	Integer.

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.security.clock.precision.lax	Specifies whether to enforce the clock precision time period. If this element is set to `true`, the client application does *not* reject SOAP responses whose time expiration period is smaller than the clock precision time, specified with the `weblogic.webservice.security.clock.precision` property. By default, the client application rejects these SOAP responses because it cannot accurately determine whether the message has expired, due to the discrepancy in clock precision between the client application and WebLogic Server. Valid values for this property are `true` and `false`. The default value is `false`.	Boolean.
weblogic.webservice.security.clock.synchronized	Specifies whether the client application assumes that the clocks of the client application invoking a WebLogic Web Service and WebLogic Server are synchronized when dealing with timestamps in SOAP messages. If the value of this property is `true`, the client application enforces, if it exists, the time expiration of the SOAP response from WebLogic Server. If the value of this element is `false`, the client application rejects all SOAP responses that contain a time expiration. Valid values for this property are `true` and `false`. The default value is `false`.	Boolean.

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.security.delay.max	Specifies, in milliseconds, the client application's expiration period for a SOAP response from WebLogic Server. The client application adds the value of this property to the creation date in the time stamp of the SOAP response, accounts for clock precision, then compares the result to the current time. If the result is greater than the current time, the client application rejects the response. In addition to its own expiration period for SOAP responses, the client application also honors expirations in the SOAP response message itself, specified by WebLogic Server To specify no expiration, set this property to `-1`. The default value of this property is `-1`. If you set this property to a value, be sure you also specify that the clocks between WebLogic Server and client applications are synchronized by setting the `weblogic.webservice.security.clock.synchronized` property to `true`.	Integer.
weblogic.webservice.security.timestamp.include	Specifies whether the client application includes a timestamp in the SOAP request to a WebLogic Web Service operation. Valid values for this property are `true` and `false`. The default value is `true`.	Boolean

Table 7-2 WebLogic Web Services System Properties

WebLogic Web Services System Property	Description	Data Type
weblogic.webservice.security.timestamp.require	Specifies whether the client application requires that the SOAP response from WebLogic Server include a timestamp. If this element is set to `true`, and a SOAP response does not contain a timestamp, the client application rejects the request. Valid values for this property are `true` and `false`. The default value is `true`.	Boolean.
weblogic.webservice.security.validity	Specifies, in milliseconds, the expiration period that the client application adds to the timestamp header of the SOAP request. To specify no expiration, set this property to `-1`. The default value of this property is `-1`.	Integer.

The following table lists the standard JDK 1.4 system properties you can set in your client applications.

For additional information about these properties, see Sun's Network Properties at http://java.sun.com/j2se/1.4/docs/guide/net/properties.html.

Table 7-3 Standard JDK 1.4 System Properties

Standard JDK 1.4 System Property	Description
http.proxyHost	If you use a proxy server to make HTTP connections, specifies the host name of the proxy server in your client applications.
http.proxyPort	If you use a proxy server to make HTTP connections, specifies the port of the proxy server in your client applications.
http.nonProxyHosts	If you use a proxy server to make HTTP connections, specifies the hosts which should be connected to directly and not through the proxy server.

Table 7-3 Standard JDK 1.4 System Properties

Standard JDK 1.4 System Property	Description
networkaddress.cache.ttl	Used in `java.security` to specify the caching policy for successful name lookups from the name service. The value is specified as an integer to indicate the number of seconds to cache the successful lookup.
networkaddress.cache.negative.ttl	Used in `java.security` to specify the caching policy for unsuccessful name lookups from the name service. The value is specified as an integer to indicate the number of seconds to cache the failure for unsuccessful lookups.
http.agent	Specifies the `User-Agent` request header sent in HTTP requests.
http.auth.digest.validateServer	Modifies the behavior of the HTTP digest authentication mechanism. When set to True, this system property forces the server to authenticate itself to the client application.
http.auth.digest.validateProxy	Modifies the behavior of the HTTP digest authentication mechanism. When set to True, this system property forces the proxy server to authenticate itself to the client application.
http.auth.digest.cnonceRepeat	Modifies the behavior of the HTTP digest authentication mechanism by specifying how many times a `cnonce` value is reused.
http.keepAlive	Specifies whether keep alive, or persistent, connections are supported.
http.maxConnections	Specifies the number of idle connections that will be simultaneously kept alive, per destination. This system property should be used together with `http.keepAlive`.
https.sharedsocket	Enables socket sharing in an SSL client application that connects to a WebLogic Web Service using the WebLogic SSL implementation. Valid values are `True` and `False`. Default value is `False`. For details, see "Using SSL Socket Sharing When Using the WebLogic SSL Implementation" on page 13-36.
https.sharedsocket.timeout	Specifies the timeout value, in seconds, for shared sockets. Default value is 15 seconds. For details, see "Using SSL Socket Sharing When Using the WebLogic SSL Implementation" on page 13-36.

Invoking Web Services from WebLogic Server

Invoking a Web Service from a component deployed on WebLogic Server, such as from an EJB or a servlet, is essentially the same as invoking a Web Service from a standalone client. You write the same code as shown in the examples in this chapter and you generate a Web Service-specific client JAR file using `clientgen` in the same way. The main differences are:

- You do not need the runtime client JAR files, because all the needed classes are already included in the WebLogic Server runtime.
- You must add the Web Service-specific client JAR file to the appropriate directory of the deployed component.
- If you are invoking a Web Service deployed on WebLogic Server Version 8.1 from a Version 7.0 instance of WebLogic Server, and it is necessary for the Version 7.0 WebLogic Server instance to use a Web Service-specific client JAR file generated from the `clientgen` Ant task included in Version 8.1 of WebLogic Server, then you must use portable stubs. For details, see "Creating and Using Portable Stubs" on page 7-22.

The following table summarizes the location of the various client JAR files depending on the type of client application from which you are invoking the Web Service.

Table 7-4 Location of Client JAR Files For Various Client Applications

Type of Client Application	Location of Runtime Client JAR Files	Location of Web Service-Specific Client JAR File
Standalone	Client's CLASSPATH.	Client's CLASSPATH
EJB	Not needed, because the runtime client classes are part of the WebLogic Server runtime.	EJB JAR file.
Servlet or Java class	Not needed, because the runtime client classes are part of the WebLogic Server runtime.	`WEB-INF/lib` directory of the WAR file.

Creating and Using Portable Stubs

If you use the Web Services client JAR files (both the ones distributed with the product and the Web Service-specific one generated by the `clientgen` Ant task) as part of an application that

runs in WebLogic Server, you might find that the Java classes in the JAR file collide with the classes of WebLogic Server itself. This happens when the WebLogic Server instance in which the client JAR file is deployed is a different version from that which the client JAR file was generated. To solve this problem, use portable stubs.

Note: Always try to use the `clientgen` Ant task of the WebLogic Server instance that is invoking the Web service to create the Web Service-specific client JAR file rather than that of the WebLogic Server that is hosting the Web Service. If this is not possible, then use portable stubs.

You need to use portable stubs only if your client application is deployed and running on WebLogic Server, not if your client application is standalone.

To enable your client application to use portable stubs:

1. Use the WebLogic Server release-specific client JAR file called `wsclient81.jar` (distributed with WebLogic Server in the *`WL_HOME`*`/server/lib` directory) with your client application rather than the generic `webserviceclient.jar` client JAR file. The `wsclient81.jar` file contains the same class files as the standard client JAR file, but they are renamed `weblogic81.*`. Because these class files are version-specific, they will not collide with any `weblogic.*` WebLogic Server classes.
2. Run the Web-service specific client JAR file you generated with the `clientgen` Ant task, as well as any supporting client JAR files, through the `VersionMaker` utility. This utility makes the following changes to the classes in these client JAR files:
   - Renames all `weblogic.*` classes as `weblogic81.*`
   - All references to `weblogic.*` classes are changed to reference `weblogic81.*` instead.

   Use these new version-specific client JAR files with your client application.

   For details on using `VersionMaker`, see "Using the VersionMaker Utility to Update Client JAR Files" on page 7-23.

Using the VersionMaker Utility to Update Client JAR Files

Follow these steps to update your client JAR files to use version-specific WebLogic Server classes:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is

BEA_HOME\user_projects\domains\domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

On UNIX, execute the setEnv.sh command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*/user_projects/domains/domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Execute the Java utility weblogic.webservice.tools.versioning.VersionMaker, passing it the following parameters:
   - destination_dir: the destination directory that will contain the new version-specific client JAR files.
   - client_jar_file: the client JAR file, generated by the clientgen Ant task, whose class files are named weblogic.* and will be renamed weblogic81.*.
   - other_jar_files: supporting JAR files

   For example:

   ```
   java weblogic.webservice.tools.versioning.VersionMaker \
      new_directory myclient.jar supporting.jar
   ```

   In the example, the weblogic.* classes in the myclient.jar and supporting.jar client JAR files are renamed weblogic81.*, and all references to these classes updated accordingly. The new client JAR files are generated into the directory called new_directory under the current directory.

Using a Proxy Server with the WebLogic Web Services Client

You can use a proxy server to proxy requests from a WebLogic Web Services client application to a server (either WebLogic or non-WebLogic) that is hosting a Web Service. However, be sure to set *all* the following system properties in your client application:

- http.proxyHost
- http.proxyPort
- weblogic.webservice.transport.http.proxy.host
- weblogic.webservice.transport.http.proxy.port

Note: If you are using HTTPS as the transport when invoking the Web Service, replace the http in the preceding properties with https. For example, use https.proxyHost instead of http.proxyHost.

For more information on these, and other, WebLogic system properties you can set in your client application, see "Using Web Services System Properties" on page 7-14.

Additionally, if you have set up your proxy server to use proxy authentication, then you must also set the property `weblogic.net.proxyAuthenticatorClassName` in your client application to the name of the Java class that implements the `weblogic.common.ProxyAuthentication` interface, as shown in the following excerpt from a client application:

```
System.setProperty("weblogic.net.proxyAuthenticatorClassName",
"my.ProxyAuthenticator");
```

In the example, `my.ProxyAuthenticator` is a class in the client application's CLASSPATH that implements the `weblogic.common.ProxyAuthentication` interface.

The `weblogic.common.ProxyAuthentication` interface allows a client application to provide user authentication information required when tunneling WebLogic HTTP and SSL protocols through a proxy server that requires user authentication. For details on implementing this interface, see the `weblogic.common.ProxyAuthentication` Javadocs.

Writing Advanced Java Client Applications

The following sections contain examples of how to write advanced client applications:

- "Writing a Dynamic Client That Uses WSDL" on page 7-25
- "Writing a Dynamic Client That Does Not Use WSDL" on page 7-27
- "Writing a Dynamic Client That Uses Non-Built-In Data Types" on page 7-29
- "Writing a J2ME Client" on page 7-31

Writing a Dynamic Client That Uses WSDL

Assume you want to create a dynamic client application that uses built-in data types and WSDL to invoke the Web Service found at the following URL:

`http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl`

Follow these steps when writing the Java code:

1. Create a service factory using the `ServiceFactory.newInstance()` method.
2. Create a `Service` object from the factory and pass it the WSDL and the name of the Web Service you are going to invoke.

3. Create a `Call` object from the `Service`, passing it the name of the port and the operation you want to execute
4. Use the `Call.invoke()` method to actually invoke the Web Service operation.

Note: If the Web Service you are invoking from your dynamic client application uses non-built-in data types, see "Writing a Dynamic Client That Uses Non-Built-In Data Types" on page 7-29.

The following Java code shows an example of writing a dynamic client application:

```
import java.net.URL;

import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

public class Main {

  public static void main(String[] args) throws Exception {

    // Setup the global SAAJ message factory
    System.setProperty("javax.xml.soap.MessageFactory",
      "weblogic.webservice.core.soap.MessageFactoryImpl");
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    // define qnames
    String targetNamespace =
      "http://www.themindelectric.com/"
      + "wsdl/net.xmethods.services.stockquote.StockQuote/";

    QName serviceName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuoteService");

    QName portName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuotePort");

    QName operationName = new QName("urn:xmethods-delayed-quotes",
                                    "getQuote");
```

```
    URL wsdlLocation =
      new
URL("http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");

    // create service
    Service service = factory.createService(wsdlLocation, serviceName);

    // create call
    Call call = service.createCall(portName, operationName);

    // invoke the remote web service
    Float result = (Float) call.invoke(new Object[] {
      "BEAS"
    });

    System.out.println("\n");
    System.out.println("This example shows how to create a dynamic client
                       application that invokes a non-WebLogic Web Service.");
    System.out.println("The webservice used was:

http://services.xmethods.net/soap/urn:xmethods-delayed-quotes.wsdl");
    System.out.println("The quote for BEAS is: ");
    System.out.println(result);
  }

}
```

Note: When you use the `javax.xml.rpc.Call` API to create a dynamic client that uses WSDL, you cannot use the following methods in your client application:

- `getParameterTypeByName()`
- `getReturnType()`

Additionally, if you want to execute the `getTargetEndpointAddress()` method, you must have previously executed the `setTargetEndpointAddress()` method, even if the targetEndPointAddress is available in the WSDL.

Writing a Dynamic Client That Does Not Use WSDL

Dynamic clients that do not use WSDL are similar to those that use WSDL except that when the client does not use WSDL, you have to explicitly set information that would be found in the WSDL, such as the parameters to the operation, the target endpoint address, and so on.

The following example shows how to create a client application that invokes a Web Service without specifying the WSDL in the client application:

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
```

```
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

public class Main {

  public static void main(String[] args) throws Exception {
    // Setup the global JAX-RPC service factory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

    // create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    // define qnames
    String targetNamespace =
      "http://www.themindelectric.com/"
      + "wsdl/net.xmethods.services.stockquote.StockQuote/";

    QName serviceName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuoteService");

    QName portName =
      new QName(targetNamespace,
                "net.xmethods.services.stockquote.StockQuotePort");

    QName operationName = new QName("urn:xmethods-delayed-quotes",
                                    "getQuote");

    // create service
    Service service = factory.createService(serviceName);

    // create call
    Call call = service.createCall();

    // set port and operation name
    call.setPortTypeName(portName);
    call.setOperationName(operationName);
    // add parameters
    call.addParameter("symbol",
                      new QName("http://www.w3.org/2001/XMLSchema", "string"),
                      ParameterMode.IN);

   call.setReturnType(new QName( "http://www.w3.org/2001/XMLSchema","float") );

    // set end point address
    call.setTargetEndpointAddress("http://www.xmethods.com:9090/soap");

    // invoke the remote web service
    Float result = (Float) call.invoke(new Object[] {
```

```
      "BEAS"
    });

    System.out.println("\n");
    System.out.println("This example shows how to create a dynamic client
                        application that invokes a non-WebLogic Web Service.");
    System.out.println("The webservice used was:

http://www.themindelectric.com/wsdl/net.xmethods.services.stockquote.StockQuot
e");
    System.out.println("The quote for BEAS is:");
    System.out.println(result);
  }

}
```

Note: In dynamic clients that do not use WSDL, the `getPorts()` method always returns `null`. This behavior is different from dynamic clients that do use WSDL in which the method actually returns the ports.

Writing a Dynamic Client That Uses Non-Built-In Data Types

When you write a dynamic client to invoke a Web Service that uses non-built-in data types as parameters or return type, you must do the following:

- Code the serialization class that converts the non-built-in data type between its Java and XML Schema representations. To create the serialization class, use the `autotype` Ant task (see "autotype" on page B-7) or create one manually (see "Writing the Serialization Class" on page 11-5.)
- In your client application code, use the JAX-RPC `TypeMappingRegistry` of the `ServiceFactory` to register the serialization classes.

For detailed information about the `TypeMappingRegistry`, see the JAX-RPC 1.0 specification at http://java.sun.com/xml/jaxrpc/index.html.

Note: Because the `clientgen` Ant task automatically generates all needed serialization classes and creates stubs that correctly use the serialization classes, BEA recommends that you use a static client application when using non-built-in data types.

The following example shows how to use the `TypeMappingRegistry` to register the serialization class called `SOAPStructCode` in your client application; the relevant code is in bold.

```
import javax.xml.soap.SOAPConstants;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
```

```
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;

import org.soapinterop.xsd.SOAPStructCodec;
import org.soapinterop.xsd.SOAPStruct;

public class MSInterop{

  public static void main( String[] args ) throws Exception{

    //set weblogic ServiceFactory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
        "weblogic.webservice.core.rpc.ServiceFactoryImpl" );

    //create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    //define qnames
    String targetNamespace = "http://soapinterop.org/";

    QName serviceName = new QName( targetNamespace, "SimpleTest" );
    QName portName = new QName( targetNamespace, "SimpleTestSoap" );

    QName operationName = new QName( "http://soapinterop.org/",
        "echoStruct" );

    //create service
    Service service = factory.createService( serviceName );

    TypeMappingRegistry registry = service.getTypeMappingRegistry();

    TypeMapping mapping = registry.getTypeMapping(
        SOAPConstants.URI_NS_SOAP_ENCODING );

    mapping.register( SOAPStruct.class,
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ),
        new SOAPStructCodec(),
        new SOAPStructCodec() );

    //create call
    Call call = service.createCall();

    //set port and operation name
    call.setPortTypeName( portName );
    call.setOperationName( operationName );
```

```
    call.addParameter( "inputStruct",
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ),
        ParameterMode.IN);

    call.setReturnType(
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ) );

    //set end point address
    call.setTargetEndpointAddress(
        "http://www.mssoapinterop.org/asmx/simple.asmx" );

    SOAPStruct s = new SOAPStruct();
    s.setVarInt(2);
    s.setVarString("foo");
    s.setVarFloat(123123);
    System.out.println(s.toString());

    SOAPStruct res = (SOAPStruct) call.invoke(new Object[]{s} );

    System.out.println( res );
  }
}
```

Writing a J2ME Client

You can create a Java 2 Platform, Micro Edition (J2ME) Web Service-specific client JAR file to use with client applications that run on J2ME.

Note: BEA supports the CDC and Foundation profile J2ME environment.

Creating a J2ME client application that invokes a Web Service is similar to creating a non-J2ME client. For example, you use the same runtime client JAR file as non-J2ME client applications (*WL_HOME*`/server/lib/webserviceclient.jar`.)

To write a J2ME client application, follow the steps described in "Creating Java Client Applications to Invoke Web Services: Main Steps" on page 7-4 but with the following changes:

- When you run the `clientgen` Ant task to generate the Web Service-specific client JAR file, be sure you specify the `j2me="True"` attribute, as shown in the following example:

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="myapps/myService_clients.jar"
           j2me="True"
/>
```

  Note: The J2ME Web Service-specific client JAR file generated by `clientgen` is not compliant with the JAX-RPC specification in the following ways:

    - The methods of the generated stubs do not throw `java.rmi.RemoteException`.
    - The generated stubs do not extend `java.rmi.Remote`.
- When you write, compile, and run your Java client application, be sure you use the J2ME virtual machine and APIs.

For more information about J2ME, see http://java.sun.com/j2me/.

Writing a J2ME Client That Uses SSL

WebLogic Server includes support for creating J2ME client applications that use SSL. If you are writing a J2ME client that uses SSL, follow these guidelines in addition to the guidelines specified in "Writing a J2ME Client" on page 7-31.

- You must use the following additional class and package:
  - `java.math.BigInteger` (class)
  - `java.util.*` (entire package)
- Ensure that the file *`WL_HOME`*`/server/lib/webserviceclient+ssl_pj.jar` is in your CLASSPATH.

  Warning: Do not include the `weblogic.jar` file in your CLASSPATH.
- If your client application uses the WSDL file to invoke a Web Service, you must use a local copy of the WSDL file stored on your client computer; you cannot access the WSDL file using a `URLConnection` object.

CHAPTER 8

Using the WebLogic Web Services APIs

The following sections provide information about using the WebLogic Web Services APIs:

- "Overview of the WebLogic Web Service APIs" on page 8-1
- "Registering Data Type Mapping Information in a Dynamic Client" on page 8-2
- "Accessing HttpSession Information from a Web Service Component" on page 8-5
- "Introspecting the WSDL from a Client Application" on page 8-6

Overview of the WebLogic Web Service APIs

WebLogic Web Services provide APIs, in addition to the JAX-RPC and SAAJ APIs, that you can use for a variety of tasks. These APIs are mostly for use in client applications, although in some cases you can use the APIs in server-side components, such as EJBs.

The sections here discuss the main usage scenarios for WebLogic Web Services APIs; for full reference information about the APIs, see the Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

The following table lists all the WebLogic Web Services APIs and their main uses.

Table 8-1 WebLogic Web Service APIs

Package Name	Typical Use Cases
`weblogic.webservice.async`	Used in a client application to invoke an operation asynchronously by splitting the invocation into two methods: the first method invokes the operation with the required parameters but does not wait for the result; later, the second method returns the actual results. See "Writing an Asynchronous Client Application" on page 7-11.
`weblogic.webservice.binding`	Used in a client application to set verbose mode and the character set programmatically. See "Setting Verbose Mode Programatically" on page 20-5 and "Setting the Character Set When Invoking a Web Service" on page 14-4.
`weblogic.webservice.client`	Used in a client application to configure SSL. See "Configuring SSL for a Client Application" on page 13-32.
`weblogic.webservice.context`	Used to access the `HttpSession` information from within the implementation of a WebLogic Web Service operation. See "Accessing HttpSession Information from a Web Service Component" on page 8-5.
`weblogic.webservice.encoding`	Used in a dynamic client application to register data type mapping information about any non-built-in data types used as a parameter or return value in a Web Service operation invocation. See "Registering Data Type Mapping Information in a Dynamic Client" on page 8-2.
`weblogic.webservice.extensions`	Used in a client application to introspect the WSDL of the invoked Web Service. See "Introspecting the WSDL from a Client Application" on page 8-6.

Registering Data Type Mapping Information in a Dynamic Client

The stubs in the client JAR file generated by the `clientgen` Ant task include the Java code needed by client applications to handle the data type mapping and registration for non-built-in data types used as parameters or return values of Web Service operations. If, however, you are

writing a dynamic client to invoke the Web Service, then you must handle the data type mapping registration yourself because dynamic clients do not use the `clientgen`-generated stubs.

The standard way to register a data type mapping is to use the JAX-RPC `TypeMappingRegistry` class. However, using this class can be cumbersome because you must register the serialization classes manually, and some of these serialization classes might be internal WebLogic classes. The following example shows how to use the `TypeMappingRegistry` class, with the relevant code in bold:

```
package examples.jaxrpc.call3;

import javax.xml.soap.SOAPConstants;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;

import org.soapinterop.xsd.SOAPStructCodec;
import org.soapinterop.xsd.SOAPStruct;

public class MSInterop{

public static void main( String[] args ) throws Exception{

    //set weblogic ServiceFactory
    System.setProperty( "javax.xml.rpc.ServiceFactory",
        "weblogic.webservice.core.rpc.ServiceFactoryImpl" );

    //create service factory
    ServiceFactory factory = ServiceFactory.newInstance();

    //define qnames
    String targetNamespace = "http://soapinterop.org/";

    QName serviceName = new QName( targetNamespace, "SimpleTest" );
    QName portName = new QName( targetNamespace, "SimpleTestSoap" );

    QName operationName = new QName( "http://soapinterop.org/",
        "echoStruct" );

    //create service
    Service service = factory.createService( serviceName );

    TypeMappingRegistry registry = service.getTypeMappingRegistry();
```

```
    TypeMapping mapping = registry.getTypeMapping(
        SOAPConstants.URI_NS_SOAP_ENCODING );

    mapping.register( SOAPStruct.class,
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ),
        new SOAPStructCodec(),
        new SOAPStructCodec() );

    //create call
    Call call = service.createCall();

    //set port and operation name
    call.setPortTypeName( portName );
    call.setOperationName( operationName );

    call.addParameter( "inputStruct",
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ),
        ParameterMode.IN);

    call.setReturnType(
        new QName( "http://soapinterop.org/xsd", "SOAPStruct" ) );

    //set end point address
    call.setTargetEndpointAddress(
        "http://www.mssoapinterop.org/asmx/simple.asmx" );

    SOAPStruct s = new SOAPStruct();
    s.setVarInt(2);
    s.setVarString("foo");
    s.setVarFloat(123123);
    System.out.println(s.toString());

    SOAPStruct res = (SOAPStruct) call.invoke(new Object[]{s} );

    System.out.println( res );
  }
}
```

The WebLogic Web Services `weblogic.webservice.encoding` package facilitates the mapping registration of non-built-in data types in a dynamic client. The API defines the following two main classes:

- `DefaultTypeMapping`: Used to register XML data types and associate them with a corresponding Java data type.
- `GenericTypeMapping`: Used to associate all non-built-in XML data types to the generic Java data type `SOAPElement`

The following procedure describes how to use the `DefaultTypeMapping` class in your dynamic client application:

1. Execute the `autotype` Ant task to create the serialization class, Java class, XML Schema, and `types.xml` file for the non-built-in data types used in your Web Service. Assume for this example that the `types.xml` file is generated in the `mydir` directory.

   For details, see "Running the autotype Ant Task" on page 6-9.

2. In your client application, import the needed WebLogic Web Service API packages:

   ```
   import weblogic.webservice.encoding.GenericTypeMapping;
   import weblogic.webservice.encoding.DefaultTypeMapping;
   ```

3. Create an instance of the JAX-RPC `TypeMappingRegistry` object:

   ```
   TypeMappingRegistry registry = service.getTypeMappingRegistry();
   ```

4. Use the `DefaultTypeMapping` class to register the `types.xml` file generated by the `autotype` Ant task, as shown in the following code excerpt:

   ```
   registry.registerDefault(
       new DefaultTypeMapping( "mydir/types.xml" ) );
   ```

Accessing HttpSession Information from a Web Service Component

Use the `weblogic.webservice.context.WebServiceSession` class to access the infrastructure inside WebLogic Server that manages HTTP Sessions. Because WebLogic Web Services are not implemented with servlets, but rather with Java classes and EJBs, the HTTP session information is not directly available, so use the `weblogic.webservice.context` package to access it. See the Javadoc for `javax.servlet.http.HttpSession` for information on using `WebServiceSession`.

In your Web Service implementation, get a `WebServiceSession` object from a `WebServiceContext` object, and then use the standard `getAttribute()` and `setAttribute()` methods to get and set attributes of the session.

Note: In order for the session functionality to work, the client application must support HTTP cookies.

The following example shows a method of the Java class that implements a Web Service that uses the `weblogic.webservice.context` API. The method maintains session state between multiple invokes of the Web Service operation.

```
import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.context.ContextNotFoundException;
import weblogic.webservice.context.WebServiceSession;

....

 /*
   * Shows how to use HTTP Session to maintain session state between
   * invokes
   */
  public int maintainSessionState(){
    try{
      WebServiceContext wsContext = WebServiceContext.currentContext();
      WebServiceSession session = (WebServiceSession)wsContext.getSession();

      Integer count = (Integer)session.getAttribute( "count" );

      count = (count==null) ?
        new Integer( 0 ) : new Integer( count.intValue() + 1 );

      session.setAttribute( "count", count );
      return count.intValue();
    }catch( ContextNotFoundException e ){
      e.printStackTrace();
      return -1;
    }
  }
```

Introspecting the WSDL from a Client Application

Use the `weblogic.webservice.extensions` package in a dynamic client application to introspect the WSDL of the Web Service you are invoking. In particular, the `WLCall` interface extends the `javax.xml.rpc.Call` interface, adding functionality to:

- Get the names of all parameters of the operation you are invoking
- Get the Java data type of a particular parameter
- Get the mode of the parameter (in, out, in-out)

The following example shows how to use the `weblogic.webservice.extensions.WLCall` interface to get more information about the parameters of the operation you are invoking in a client application.

```
import java.io.IOException;

import java.net.URL;

import java.util.Iterator;
```

```
import java.rmi.RemoteException;

import javax.xml.namespace.QName;

import javax.xml.rpc.Service;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ServiceException;
import javax.xml.rpc.Call;

import javax.xml.rpc.encoding.TypeMapping;
import javax.xml.rpc.encoding.TypeMappingRegistry;

import weblogic.webservice.encoding.GenericTypeMapping;
import weblogic.webservice.extensions.WLCall;

public class BrowserClient{

  public void invoke() throws RemoteException, InvokeFailedException,
    ServiceException, IOException{

    String url = "http://localhost:7001/mega/TemperatureService?WSDL";

    System.setProperty( "javax.xml.rpc.ServiceFactory",
        "weblogic.webservice.core.rpc.ServiceFactoryImpl" );

    System.setProperty( "weblogic.webservice.servicenamechecking",
        "false" );

    ServiceFactory factory = ServiceFactory.newInstance();

    QName serviceName = new QName( "http://www.bea.com/mega-service/",
        "MegaWebService" );

    Service service = factory.createService( new URL( url ), serviceName );

    TypeMappingRegistry registry = service.getTypeMappingRegistry();
    registry.registerDefault( new GenericTypeMapping() );

    System.out.println( "+ Service: " + service.getServiceName() );

    for( Iterator it = service.getPorts(); it.hasNext(); ){
      QName portName = (QName)it.next();
      System.out.println( "  + Port: " + portName );
      Call[] calls = service.getCalls( portName );
      printCalls( calls );
    }
  }

  private void printCalls( Call[] calls ){
    for( int i=0; i<calls.length; i++ ){
      Call call = calls[i];
```

```
      System.out.println( "    + Operation :" + call.getOperationName() );
      printParameters( (WLCall)call );

      if( call.getReturnType() != null ){
        System.out.println( "      + Return Type:" + call.getReturnType() );
      }

      System.out.println( "" );
    }
  }

  private void printParameters( WLCall call ){
    for( Iterator it = call.getParameterNames(); it.hasNext(); ){
      String name = (String)it.next();
      System.out.println( "      + Part :" + name );

      System.out.println( "        - Java Type :" +
          call.getParameterJavaType( name ) );

      System.out.println( "        - Mode :" +
          call.getParameterMode( name ) );

      System.out.println( "        - XML Type :" +
          call.getParameterTypeByName( name ) );
    }
  }
}
```

CHAPTER 9

Using JMS Transport to Invoke a WebLogic Web Service

The following sections provide information about using JMS transport to invoke a WebLogic Web Service:

- "Overview of Using JMS Transport" on page 9-1
- "Specifying JMS Transport for a WebLogic Web Service: Main Steps" on page 9-2
- "Updating the web-services.xml File to Specify JMS Transport" on page 9-3
- "Invoking a Web Service Using JMS Transport" on page 9-3

Overview of Using JMS Transport

By default, client applications use HTTP/S as the connection protocol when invoking a WebLogic Web Service. You can, however, configure a WebLogic Web Service so that client applications can also use JMS as the transport when invoking the Web Service.

When a WebLogic Web Service is configured to use JMS as the connection transport:

- The generated WSDL of the Web Service contains *two* port definitions: one with an HTTP/S binding and one with a JMS binding. When you invoke the Web Service in your client application, you can choose which port, and thus which type of transport, you want to use.

  Warning: Non-WebLogic client applications, such as a .NET client, will not be able to invoke the Web Service using the JMS binding.

- The `clientgen` Ant task creates a `Service` implementation that contains *two* `getPortXXX()` methods, one for HTTP/S and one for JMS.

Note: You can configure *any* WebLogic Web Service to include a JMS binding in its WSDL. This feature is independent of JMS-implemented WebLogic Web Services.

Specifying JMS Transport for a WebLogic Web Service: Main Steps

In the following procedure, it is assumed that you are familiar with the `servicegen` Ant task and you want to update the Web Service to use JMS transport. For an example of using `servicegen`, see Chapter 3, "Creating a WebLogic Web Service: A Simple Example."

Some of the main steps include configuring JMS resources using the Administration Console.

1. Invoke the Administration Console to in your browser, as described in "Overview of Administering WebLogic Web Services" on page 17-1.
2. Use the Administration Console to create (if they do not already exist) and configure the following JMS components of WebLogic Server:
   - JMS Template
   - JMS Connection factory
   - JMS Server

     When creating the JMS Server, or configuring an existing one, be sure that you specify the JMS Template you already created for the Temporary Template attribute.

     Note: Do not target this JMS Server to a Migratable Target.
   - JMS Queue (associated with the preceding JMS Server).

   For details about creating all these components, see *JMS: Configuring* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html#jms_queue_create.
3. Update the `web-services.xml` file of your WebLogic Web Service to specify that the generated WSDL include a port that uses a JMS binding.

   See "Updating the web-services.xml File to Specify JMS Transport" on page 9-3.
4. Redeploy the Web Service.

See "Invoking a Web Service Using JMS Transport" on page 9-3 for details about writing a Java client application that invokes your Web Service.

Updating the web-services.xml File to Specify JMS Transport

The `web-services.xml` file is located in the `WEB-INF` directory of the Web application of the Web Services EAR file. See "The Web Service EAR File Package" on page 6-16 for more information on locating the file.

To update the `web-services.xml` file to specify JMS transport, follow these steps:

1. Open the file in your favorite editor.
2. Add the `jmsUri` attribute to the `<web-service>` element that describes your Web Service and set the attribute to the following value:

   `connection-factory-name/queue-name`

   where *`connection-factory-name`* and *`queue-name`* are the JNDI names of the JMS connection factory and JMS queues, respectively, that you previously created. For example:

```
<web-service
   name="myJMSTransportWebService"
   jmsUri="JMSTransportFactory/JMSTransportQueue"
   ...>
...
</web-service>
```

Invoking a Web Service Using JMS Transport

Invoking a WebLogic Web Service using the JMS transport is very similar to using HTTP/S, as described in Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server," but with a few differences, as described in the following procedure.

1. Re-run the `clientgen` Ant task.

   Because the WSDL of the Web Service has been updated to include an additional port with a JMS binding, the `clientgen` Ant task automatically creates new stubs that contains these JMS-specific `getPortXXX()` methods.

   For details, see "Generating the Client JAR File by Running the clientgen Ant Task" on page 7-5.

2. Update the CLASSPATH of your client application to include the standard JMS client JAR files:

```
WL_HOME/server/lib/wlclient.jar
WL_HOME/server/lib/wljmsclient.jar
```

where *WL_HOME* refers to the main WebLogic Server installation directory.

For more information on JMS client JAR files, see *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs81/jms/index.html.

3. Update your client application to use the new `getPortXXX()` method of the JAX-RPC `Service` class generated by the `clientgen` Ant task. The standard `getPortXXX()` method for HTTP/S is called `get`*ServiceName*`Port()`; the new method to use the JMS transport is called `get`*ServiceName*`PortJMS()`, where *ServiceName* refers to the name of your Web Service. These two `gerPortXXX()` methods correspond to the two port definitions in the generated WSDL of the Web Service, as described in "Overview of Using JMS Transport" on page 9-1.

   The following example of a simple client application shows how to invoke the `postWorld` operation of the `MyService` Web Service using both the HTTP/S transport (via the `getMyservicePort()` method) and the JMS transport (via the `getMyServicePortJMS()` method):

```
package examples.jms.client;

import java.io.IOException;

public class Main{

  public static void main( String[] args ) throws Exception{

    MyService service = new MyService_Impl();

    { //using HTTP transport
      MyServicePort port = service.getMyServicePort();
      port.postWorld( "using HTTP" );
    }

    { //using JMS transport
      MyServicePort port = service.getMyServicePortJMS();
      port.postWorld( "using JMS" );
    }
  }
}
```

CHAPTER 10

Using Reliable SOAP Messaging

The following sections describe how to use reliable SOAP messaging, both as a sender and a receiver of a SOAP message, between WebLogic Server instances:

- "Overview of Reliable SOAP Messaging" on page 10-1
- "Using Reliable SOAP Messaging: Main Steps" on page 10-6

Warning: Reliable SOAP Messaging is not supported in a clustered environment.

Overview of Reliable SOAP Messaging

Reliable SOAP messaging is a framework whereby an application running in one WebLogic Server instance can asynchronously and reliably invoke a Web Service running on another WebLogic Server instance. *Reliable* is defined as the ability to guarantee message delivery between the two Web Services.

Note: Reliable SOAP messaging works between two Web Services deployed on a single WebLogic Server instance. Typically this setup is used for development. However, in real-life, reliable SOAP messaging is meant to be used between *two* WebLogic Server instances, both of which must be configured to use reliable SOAP messaging.

The sender WebLogic Server has an application that asynchronously invokes a reliable Web Service operation running on the receiver WebLogic Server. The sender sends the receiver a SOAP message that has reliable SOAP messaging information in its header. The Web Service operation being invoked has been configured for reliable SOAP messaging. Due to the asynchronous nature of the invoke, the sender does not immediately know whether the relevant

operation has been invoked, but it has the guarantee that it will get one of two possible notifications:

- The message has been received by the receiver.

  Note: This does not mean that the Web Service operation on the receiver WebLogic Server was invoked *successfully*; the operation might fail due to an application exception. The exception is included in the notification to the sender. For details about transactions, see "Receiver Transactional Context" on page 10-4.

- The sender was unable to deliver the message.

Using the Weblogic Web Services asynchronous API, the sender can either poll the receiver for notification, or register a callback to be notified. Eventually, either the sender receives a notification that the message was received, or it receives notification that the message was not delivered.

Reliable SOAP messaging is transport-independent. By default, it uses HTTP/S. However, you can also use JMS if you configure the receiving Web Service appropriately and use the JMS port when the sender invokes the Web Service. For details on using JMS transport, see Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service."

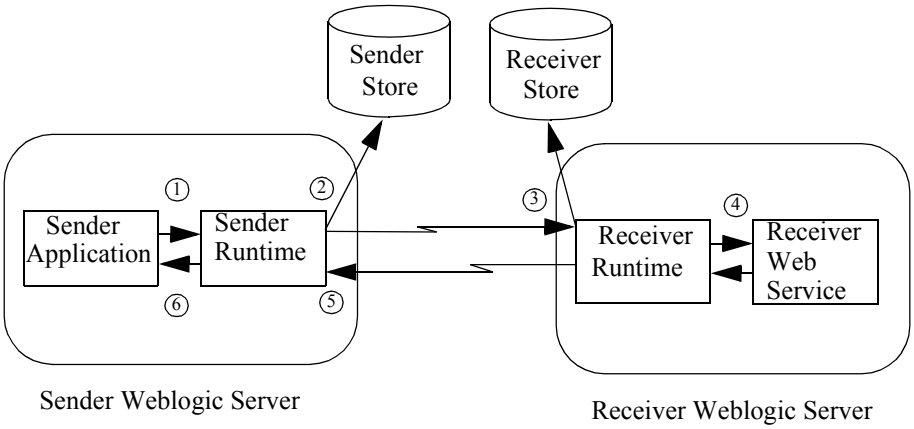
Reliable SOAP Messaging Architecture

The following terms are used in this section:

- *sender*: The WebLogic Server instance that sends the reliable SOAP message.
- *sender application*: The user application running in the sender that reliably invokes a Web Service operation running on the receiver.
- *sender runtime*: The WebLogic Server code running on the sender that handles reliable SOAP messaging.
- *receiver*: The WebLogic Server instance that receives a reliable SOAP message.
- *receiver Web Service*: The Web Service running on the receiver that contains the operation configured to be invoked reliably.
- *receiver runtime*: The WebLogic Server code running on the receiver that handles reliable SOAP messaging.

The following diagram and corresponding steps describe the architecture of the reliable SOAP messaging feature.

Figure 10-1 Reliable SOAP Messaging Architecture



1. The sender application invokes a reliable operation running on the receiver WebLogic Server.
2. The sender runtime saves the message in its persistent JMS store. The store can be either a JMS File or JDBC store.

   The sender runtime sends the SOAP message to the receiver WebLogic Server.
3. The receiver runtime receives the message, checks for duplicates in its persistent JMS store, and if none are found, saves the message ID in its store. If it finds a duplicate, the receiver runtime acknowledges the message, but does not deliver it to the receiver Web Service.

   Note: Only the message ID, and not the entire message itself, is persisted in the receiver's store.

   The actions performed by the receiver execute within the context of a transaction. See "Receiver Transactional Context" on page 10-4.
4. The receiver runtime invokes the reliable operation and sends an acknowledgement back to the sender in the SOAP header.

   Because only `void` operations can be invoked reliably, the receiver does not return any values to the sender. If the invoked operation throws an application exception, the exception *is*, however, sent back to the sender. System exceptions (from EJBs, but not from Java classes) roll back the transaction started by a receiver. For details, see "Receiver Transactional Context" on page 10-4.
5. The sender runtime removes the message from its persistent store so that the message does not get sent again.

The sender is configured to retry sending the message if it does not receive notification of receipt. You configure the number of retries, and amount of time between retries, of the sender using the Administration Console. Once sender runtime has resent the message the maximum number of retries, it removes the message from its store.

6. The sender runtime sends notification to the sender application (either via callbacks or polling) that either the message was received and the operation invoked or that it was never successfully delivered.

Receiver Transactional Context

When the receiver runtime receives a message from a sender, it starts a transaction, and the following subsequent receiver actions execute within the context of this transaction:

1. Receives a message from the sender.
2. Starts a transaction.
3. Checks for duplicates in its persistent store.
4. If duplicates are found, the receiver sends an acknowledgement back to the sender and rolls back the transaction.
5. If no duplicates are found, saves the message ID in its store.
6. Invokes the operation.
7. Sends an acknowledgment back to the sender.
8. Commits the transaction.

The main reason the receiver executes all its actions within the context of a transaction is to preserve the integrity of the message IDs in its persistent store. For example, suppose WebLogic Server crashes right after the receiver saves a message ID in its store, but before it invokes the operation; in this case, the transaction is rolled back and the saved message ID is removed from the store. Later, when the sender resends the message (because it has not yet received an acknowledgement that the operation was invoked), the receiver has no history of the message and will correctly go through the whole process again. If the receiver had not executed within the context of a transaction, it would never invoke the operation in this case because of the incorrect presence of the message ID in its store.

The transaction started by the receiver is rolled back if any of the following events occurs during the transaction:

- WebLogic Server crashes.
- The EJB container or EJB application method that implements the operation issues a rollback.
- The Java class method that implements the operation issues a rollback.
- The EJB container or EJB application method throws a system exception, such as a `RemoteException`.

The following events do not cause a rollback of the transaction:

- The EJB application method throws an application exception. An example of an application exception is `WithdrawalErrorException`, which is thrown by a method when a user tries to withdraw too much money from their account.
- The Java class method throws *any* exception.

Guidelines For Programming the EJB That Implements a Reliable Web Service Operation

When creating a stateless session EJB-implemented Web Service whose operations can be invoked reliably, follow these guidelines when programming the EJB:

- The EJB must use only container-managed transactions; bean-managed transactions are not supported.
- The transactional attribute of the EJB method that implements the reliable operation must be set to one of the following values:
  - `Required` (recommended)
  - `Supports`
  - `Mandatory`

  Set the transactional attribute of an EJB method with the `<trans-attribute>` element in the `ejb-jar.xml` deployment descriptor.
- If you want to explicitly roll back the transaction from the EJB application method, use the `EJBContext.setRollbackOnly()` method.
- Be aware that system exceptions (such as `RemoteException`) thrown by the EJB container or the EJB application method will roll back the transaction. Application exceptions (such as `WithdrawalErrorException` thrown by the EJB when a user tries to withdraw too much money from an account) do not roll back the transaction.

For more information, see Programming WebLogic Enterprise JavaBeans at http://e-docs.bea.com/wls/docs81/ejb/index.html and Programming WebLogic JTA at http://e-docs.bea.com/wls/docs81/jta/index.html.

Guidelines for Programming the Java Class That Implements a Reliable Web Service Operation

When creating a Java class-implemented Web Service whose operations can be invoked reliably, follow these guidelines when programming the Java class:

- If you want to roll back the transaction from the Java method, use the Java Transaction API (JTA) to get the transaction object and then explicitly roll back the transaction.
- Be aware exceptions thrown by the Java class never roll back the transaction.

For more information, see Programming WebLogic JTA at http://e-docs.bea.com/wls/docs81/jta/index.html.

Configuring the Transaction

Use the Administration Console to configure the following transaction attributes:

- Transaction time-out and limits
- Transaction manager behavior

Configuration settings for JTA transactions are applicable at the domain level. This means that configuration attribute settings apply to all servers within a domain. For details, see Configuring Transactions at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jta.html.

Limitations of Reliable SOAP Messaging

- Only Web Service operations that return `void` can be configured to be invoked reliably.
- One-way Web Service operations cannot be configured to be invoked reliably.

Using Reliable SOAP Messaging: Main Steps

The following procedure describes the main steps for configuring reliable SOAP messaging to invoke a WebLogic Web Service operation. The procedure describes configuration and code-writing tasks that take place in both the sender and receiver WebLogic Server instances.

Note: It is assumed that you have already implemented and assembled a WebLogic Web Service and you want to enable one or more of its operations to be invoked reliably. Additionally, it is assumed that you have already coded a server-side application (such as a servlet in a Web application) that invokes the Web Service in a non-reliable way and you want to update the application to invoke the Web Service reliably.

For details about these tasks, see Chapter 5, "Implementing WebLogic Web Services," Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks," and Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server."

1. Configure the reliable SOAP messaging attributes for the *sender* WebLogic Server instance.

   See "Configuring the Sender WebLogic Server" on page 10-8.

2. Configure the reliable SOAP messaging attributes for the *receiver* WebLogic Server instance.

   See "Configuring the Receiver WebLogic Server" on page 10-10.

3. Update the `build.xml` file that contains the call to the `servicegen` Ant task, adding the `<reliability>` child element to the `<service>` element that builds your Web Service on the *receiver* WebLogic Server, as shown in the following example:

```
<servicegen
   destEar="ears/myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="jars/myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True" >
       <reliability duplicateElimination="True"
                    persistDuration="60"
       />
   </service>
</servicegen>
```

   In the example, the Web Service ignores duplicate invokes from the same sender application and persists messages for at least 60 seconds. For more information on the attributes of the `<reliability>` element, see "servicegen" on page B-25.

Note: When you regenerate your Web Service using this `build.xml` file, *every* operation that returns `void` will be enabled for reliable invocation. If you want only certain operations to be invoked reliably, or you prefer not to regenerate your Web Service using `servicegen`, you can update the `web-services.xml` file of your WebLogic Web Service manually. For details, see "Updating the web-services.xml File Manually for Reliable SOAP Messaging" on page 10-15.

4. Re-run the `servicegen` Ant task to regenerate your receiver Web Service.
5. Re-run the `clientgen` Ant task, specifying the `generateAsyncMethods="True"` attribute, to generate a new Web Service-specific client JAR file that contains the asynchronous operation invocations. This new client JAR file will be used with the server-side application running in the sender WebLogic Server.
6. On the client application running on the sender WebLogic Server, update the Java code that invokes the Web Service to invoke it reliably.

   For an example, see "Writing the Java Code to Invoke an Operation Reliably" on page 10-11.

Configuring the Sender WebLogic Server

This section describes how to configure reliable SOAP messaging attributes for a WebLogic Server instance in its role as a sender of a reliable SOAP message.

Note: Part of the reliable SOAP messaging configuration involves configuring, if it does not already exist, a JMS File or JDBC store.

The following table describes the reliable SOAP messaging attributes.

Table 10-1 Reliable SOAP Messaging Attributes for a Sender WebLogic Server

Attribute	Description
Store	The persistent JMS store used by WebLogic Server, in its role as a sender, to persist the reliable SOAP messages that it sends.

Table 10-1 Reliable SOAP Messaging Attributes for a Sender WebLogic Server

Attribute	Description
Default Retry Count	The default maximum number of times that the sender runtime should attempt to re-deliver a message that the receiver WebLogic Server has not yet acknowledged. Default value is 10.
Default Retry Interval	The default minimum number of seconds that the sender runtime should wait between retries if the receiver does not send an acknowledgement of receiving the message, or if the sender runtime detects a communications error while attempting to send a message. Default value is 600.

To configure these attributes:

1. Invoke the Administration Console in your browser, as described in "Overview of Administering WebLogic Web Services" on page 17-1.
2. Create, if one does not already exist, a JMS store. This can be either a JMS File store or a JMS JDBC store. See *JMS File Store Tasks* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html#configure_jms_file_stores and *JMS JDBC Store Tasks* at *http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html#configure_jms_jdbc_stores* .

   Warning: The JMS Store is not migratable.
3. Click the Servers node in the left pane.
4. Select the WebLogic Server for which you want to configure reliable SOAP messaging in its role as a sender.
5. In the right pane, select the Services→Web Services tab.
6. Select the JMS store from the Store drop-down list that will contain WebLogic Server's reliable SOAP messages when acting as a sender.
7. Enter the default maximum number of times the sender WebLogic Server should attempt to resend a message in the Default Retry Count field.

8. Enter the default minimum number of seconds that the sender WebLogic Server should wait between retries in the Default Retry Interval field.
9. Enter the default minimum number of seconds that the receiver of the reliable SOAP message should persist the history of the message in its JMS store in the Default Time to Live field

   Warning: This value should be larger than the corresponding value of any Web Service operation being invoked reliably. Later sections describe how to configure his value in the Web Service's `web-services.xml` file by updating the `persist-duration` attribute of the `<reliable-delivery>` subelement of the invoked `<operation>`.

10. Click Apply.

Configuring the Receiver WebLogic Server

This section describes how to configure reliable SOAP messaging attributes for a WebLogic Server instance in its role as a receiver of a reliable SOAP message.

Note: Part of the reliable SOAP messaging configuration involves configuring, if it does not already exist, a JMS File or JDBC store.

The following table describes the reliable SOAP messaging attributes.

Table 10-2 Reliable SOAP Messaging Attributes for a Receiver WebLogic Server

Attribute	Description
Store	The persistent JMS store used by the receiver WebLogic Server to persist the history of a reliable SOAP message sent by a sender.
Default Time To Live	The default number of seconds that the receiver of the reliable SOAP message should persist the history of the message in its store. If the Default Time to Live number of seconds have passed since the message was first sent, the sender will not resend a message with the same message ID. If a sender cannot send a message successfully before the Default Time To Live number of seconds has passed, the sender reports a delivery failure.

To configure these attributes:

1. Invoke the Administration Console in your browser, as described in "Overview of Administering WebLogic Web Services" on page 17-1.
2. Create, if one does not already exist, a JMS store. This can be either a JMS File store or a JMS JDBC store. See *JMS File Store Tasks* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html#configure_jms_file_stores and *JMS JDBC Store Tasks* at *http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html#configure_jms_jdbc_stores* .

   Warning: The JMS Store is not migratable.
3. Click the Servers node in the left pane.
4. Select the WebLogic Server for which you want to configure reliable SOAP messaging in its role as a receiver.
5. In the right pane, select the Services→Web Services tab.
6. Select the JMS store from the Store drop-down list that will be used for duplicate elimination by the receiver.
7. Enter the number of seconds in the Default Time to Live field.

   Later sections in this document describe how each Web Service operation can override this default value. See "Updating the web-services.xml File Manually for Reliable SOAP Messaging" on page 10-15.
8. Click Apply.

Writing the Java Code to Invoke an Operation Reliably

You specify that a WebLogic Web Service operation is reliable by updating its definition in the `web-services.xml` file, adding the `<reliable-deliver>` child element to the corresponding `<operation>` element. You can do this using the `servicegen` Ant task (see "Using Reliable SOAP Messaging: Main Steps" on page 10-6), or by updating the `web-services.xml` file manually (see "Updating the web-services.xml File Manually for Reliable SOAP Messaging" on page 10-15). A client application, however, is not *required* to invoke a reliable operation in a reliable manner. There are three ways to invoke a reliable operation:

- Synchronously with no reliability. This is the standard JAX-RPC way of invoking operations.

- Asynchronously with no reliability, as described in "Writing an Asynchronous Client Application" on page 7-11.
- Asynchronously with reliability, as described in this chapter.

Writing the Java code to invoke a Web Service operation reliably from a sender application is very similar to invoking an operation asynchronously, as described in "Writing an Asynchronous Client Application" on page 7-11. The asynchronous invoke of an operation is split into two methods: `start`*`Operation`*`()` and `end`*`Operation`*`()`.

In addition to the standard asynchronous client Java code, to invoke an operation reliably you must:

- Enable reliable delivery in your client application with the `AsyncInfo.setReliableDelivery(true)` method.

  This method also checks for correct JMS configuration and throws an exception if it finds any errors in the configuration.

- Optionally create and set a listener to listen for the results of a reliable operation invocation with the `AsyncInfo.setResultListener(listener)` method. The listener class implements the `ResultListener` interface, which in turn defines the `onCompletion()` listener callback method in which you define what happens when the asynchronous reliable operation invocation completes.

The following example shows the simplest way to invoke the `processOrder()` operation asynchronously and reliably by specifying the `setReliableDeliver(true)` method and using the asynchronous API to split the operation into two invocations: `startProcessOrder()` and `endProcessOrder()`. You tell the `clientgen` Ant task to generate these two methods in the stubs by specifying the `generateAsyncMethods` attribute.

```
import weblogic.utils.Debug;

import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.FutureResult;

public final class ReliableSender {

public void placeOrder(String order) {
   try {
      // set up Web Service port
      MarketService market = new MarketService_Impl();
      MarketServicePort marketPort = marketService.getMarketServicePort();

      // enable reliable delivery
      AsyncInfo asyncCtx = new AsyncInfo();
      asyncCtx.setReliableDelivery(true);
```

```
      // call the Web Service asynchronously
     FutureResult futureResult = marketPort.startProcessOrder(order, asyncCtx);
      marketPort.endProcessOrder(futureResult);

    } catch (Exception e) {
      Debug.say("Exception in ReliableSender: " + e);
    }
  }
}
```

The following more complex example builds on the previous by setting a result listener to listen for the completion of the asynchronous and reliable operation invoke. The implementation of the `onCompletion()` method specifies what happens when the invoke completes; in the example, a message is printed if the invoke failed.

```
import java.io.Serializable;

import weblogic.webservice.async.AsyncInfo;
import weblogic.webservice.async.FutureResult;
import weblogic.webservice.async.InvokeCompletedEvent;
import weblogic.webservice.async.ResultListener;
import weblogic.webservice.async.ReliableDeliveryFailureEvent;

import weblogic.utils.Debug;

public final class ReliableSender {

  public void placeOrder(String order) {
    try {
      // set up Web Service port
      MarketService market = new MarketService_Impl();
      MarketServicePort marketPort = marketService.getMarketServicePort();

      // enable reliable delivery
      AsyncInfo asyncCtx = new AsyncInfo();
      asyncCtx.setReliableDelivery(true);

      // set up the result listener
      RMListener listener = new RMListener();
      asyncCtx.setResultListener(listener);

      // call the Web Service asynchronously
     FutureResult futureResult = marketPort.startProcessOrder(order, asyncCtx);

      while ( !futureResult.isCompleted()) {
        Debug.say("async polling  ...");  // do something else
        Thread.sleep(3000);
      }

      marketPort.endProcessOrder(futureResult);
```

```
    } catch (Exception e) {
      Debug.say("Exception in ReliableSender: " + e);
    }
  }
}

class RMListener implements ResultListener, Serializable {

  public void onCompletion(InvokeCompletedEvent event) {
    if (event instanceof ReliableDeliveryFailureEvent) {
      ReliableDeliveryFailureEvent rdEvent =
          (ReliableDeliveryFailureEvent) event;
      Debug.say("Reliable delivery failed with the following message: " +
                rdEvent.getErrorMessage());
    }
  }
}
```

Handling Sender Server Failures

The application that invokes an operation reliably must handle the case where the sender server crashes in the middle of retrying SOAP message delivery. Once the sender server restarts, it will check its persistent store for any messages that have not yet been successfully delivered, and if it finds any, it will continue trying to send the message to the receiver server. The problem, however, is that due to the sender server crash, the application that initially invoked the operation reliably may not be deployed anymore, and when the receiver server sends back an acknowledgement after the sender server restarts, there will be no application to accept the acknowledgment.

To handle this situation correctly, code your application to follow these guidelines:

- Create a class that implements the `ResultListener` interface. This class listens for the completion of the reliable operation invoke. See the second example "Writing the Java Code to Invoke an Operation Reliably" on page 10-11 for a sample of writing this class.
- Code the class that implements the `ResultListener` interface to also implement the `Serializable` interface to ensure that, in case of a sender server crash, the class will be serialized and stored on disk. Then, once the sender server restarts, the result listener class will also be invoked and will handle subsequent acknowledgment messages from the receiver.
- Be sure to also serialize any information needed by the result listener class so that once the class is instantiated after a sender server crash it can return to its previous state and correctly handle acknowledgments from the receiver.

Updating the web-services.xml File Manually for Reliable SOAP Messaging

If you regenerated your Web Service using the `servicegen` Ant task, *every* operation that returns `void` is enabled for reliable invocation. If you want only certain operations to be invoked reliably, or you prefer not to regenerate your Web Service using `servicegen`, you can update the `web-services.xml` file of your WebLogic Web Service manually, as described in this section.

The `web-services.xml` file is located in the `WEB-INF` directory of the Web application of the Web Services EAR file. See "The Web Service EAR File Package" on page 6-16 for more information on locating the file.

To update the `web-services.xml` file to enable reliable SOAP messaging for one or more operations:

1. Open the file in your favorite editor.
2. For each operation for which you want to enable reliable SOAP messaging, add a `<reliable-delivery>` subelement and specify the following optional attributes:
   - `duplicate-elimination` - Boolean that specifies whether the WebLogic Web Service should ignore duplicate invokes with the same message ID from the same sender application. Default value is `True`.
   - `persist-duration` - Integer value that specifies the minimum number of seconds that the Web Service should persist the history of the reliable SOAP message (received from the sender that invoked the Web Service) in its storage. When persist-duration seconds have elapsed, the receiver WebLogic Server deletes the history of the message from its store. The value of this attribute, if you set it, should be greater than the product of the retry interval and the retry count of the sender.

     This attribute overrides the default server value you set in "Configuring the Receiver WebLogic Server" on page 10-10. The default if neither is set is 360 seconds.

   The following example shows an operation that can be invoked reliably:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
    <reliable-delivery persist-duration="80" />
</operation>
```

CHAPTER 11

Using Non-Built-In Data Types

The following sections describe how to use non-built-in data types in WebLogic Web Services:

- "Overview of Using Non-Built-In Data Types" on page 11-1
- "Creating Non-Built-In Data Types Manually: Main Steps" on page 11-2

Overview of Using Non-Built-In Data Types

You can create a WebLogic Web Service that uses non-built-in data types as the Web Service parameters and return value. Non-built-in data types are defined as data types other than the supported built-in data types, such as `int` and `String`. For the full list of built-in types, see "Supported Built-In Data Types" on page 5-15.

WebLogic Server transparently handles the conversion of the built-in data types between their XML and Java representation. However, if your Web Service operation uses non-built-in data types, you must provide the following information so that WebLogic Server can perform the conversion:

- Serialization class that converts between the XML and Java representation of the data.
- A Java class to contain the Java representation of the data type.
- An XML Schema representation of the data type.
- Data type mapping information in the `web-services.xml` deployment descriptor file.

WebLogic Server includes the `servicegen` and `autotype` Ant tasks which automatically generate the preceding components by introspecting the stateless session EJB or Java class

back-end component for your Web Service. These Ant tasks can handle many non-built-in data types, so most programmers will not ever have to create the components manually.

Sometimes, however, you may need to create the non-built-in data type components manually. Your data type may be so complex that the Ant task cannot correctly generate the components. Or maybe you want more control over how the data is converted between its XML and Java representations rather than relying on the default conversion procedure used by WebLogic Server.

For a full list of the supported non-built-in data types, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.

For procedural instructions on using `servicegen` and `autotype`, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks." For reference information, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

Creating Non-Built-In Data Types Manually: Main Steps

The following procedure describes how to create non-built-in data types and use the `servicegen` Ant task to create a deployable Web Service:

1. Write the XML Schema representation of your data type. See "Writing the XML Schema Data Type Representation" on page 11-3.
2. Write a Java class that represents your data type. See "Writing the Java Data Type Representation" on page 11-4.
3. Write a serialization class that converts the data between its XML and Java representations. See "Writing the Serialization Class" on page 11-5.
4. Compile your Java code into classes. Ensure that your CLASSPATH variable can locate the classes.
5. Create a text file that contains the data type mapping information about your non-built-in data type. See "Creating the Data Type Mapping File" on page 11-10.
6. Assemble your Web Service using the `servicegen` Ant task as described in "Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3, with the following addition: when creating the `build.xml` file that calls the `servicegen` Ant task, be sure you specify the `typeMappingFile` attribute of `servicegen`, setting it equal to the name of the data type mapping file you created in the preceding step.

   BEA recommends that you create an exploded directory, rather than an EAR file, by specifying a value for the `destEar` attribute of `servicegen` that does *not* have an `.ear`

suffix. You can later package the exploded directory into an EAR file when you are ready to deploy the Web Service.

7. Update the `web-services.xml` file (which was generated by the `servicegen` Ant task), adding the XML Schema representation of your data type that you created in the first step of this procedure. See "Updating the web-services.xml File With XML Schema Information" on page 11-11.
8. Either deploy the exploded directory as your Web Service, or package the directory into an EAR file and deploy it on WebLogic Server.
9. If you want to use the `clientgen` Ant task to generate a Java client, follow the procedure described in "Running the clientgen Ant Task" on page 6-12 with the following additions to the `build.xml` file that calls `clientgen`:
   - Specify the `ear` attribute and set it to the full name of your Web Service EAR file. Do *not* specify the `wsdl` attribute.
   - Specify the `useServerTypes` attribute and set it to `True`.

Writing the XML Schema Data Type Representation

Web Services use SOAP as the message format to transmit data between the service and the client application that invokes the service. Because SOAP is an XML-based protocol, you must use XML Schema notation to describe the structure of non-built-in data types used by Web Service operations.

Warning: XML Schema is a powerful and complex data description language, and its use is not recommended for the faint of heart.

The following example shows the XML Schema that describes a non-built-in data type called `EmployBean`:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:stns="java:examples.newTypes"
            attributeFormDefault="qualified"
            elementFormDefault="qualified"
            targetNamespace="java:examples.newTypes">
   <xsd:complexType name="EmployeeBean">
      <xsd:sequence>
        <xsd:element name="name"
                     type="xsd:string"
```

```
                        nillable="true"
                        minOccurs="1"
                        maxOccurs="1">
            </xsd:element>
            <xsd:element name="id"
                        type="xsd:int"
                        minOccurs="1"
                        maxOccurs="1">
            </xsd:element>
          </xsd:sequence>
       </xsd:complexType>
    </xsd:schema>
```

The following XML shows an instance of the `EmployeeBean` data type:

```
<EmployeeBean>
  <name>Beverley Talbott</name>
  <id>1234</id>
</EmployeeBean>
```

For detailed information about using XML Schema notation to describe your non-built-in data type, see the XML Schema specification at http://www.w3.org/TR/xmlschema-0/.

Writing the Java Data Type Representation

You use the Java representation of the non-built-in data type in your EJB or Java class that implements the Web Service operation.

The following example shows one possible Java representation of the `EmployeeBean` data type whose XML representation is described in the preceding section:

```
package examples.newTypes;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */

public final class EmployeeBean {

  private String name = "John Doe";
  private int id = -1;

  public EmployeeBean() {
  }
```

```
  public EmployeeBean(String n, int i) {
    name = n;
    id = i;
  }

  public String getName() {
    return name;
  }
  public void setName(String  v) {
    this.name = v;
  }

  public int getId() {
    return id;
  }
  public void setId(int  v) {
    this.id = v;
  }

  public boolean equals(Object obj) {
    if (obj instanceof EmployeeBean) {
      EmployeeBean e = (EmployeeBean) obj;
      return (e.name.equals(name) && (e.id == id));
    }
    return false;
  }
}
```

Writing the Serialization Class

The serialization class performs the actual conversion of your data between its XML and Java representations. You write only one class that contains methods to serialize and deserialize your data. In the class you use the WebLogic XML Streaming API to process the XML data.

The WebLogic XML Streaming API provides an easy and intuitive way to consume and generate XML documents. It enables a procedural, stream-based handling of XML documents.

For detailed information on using the WebLogic XML Streaming API, see *Programming WebLogic XML* at http://e-docs.bea.com/wls/docs81/xml/xml_stream.html.

For more information on the WebLogic Web Services and XML APIs used in this section, see the Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

The following example shows a class that uses the XML Streaming API to serialize and deserialize the data type described in "Writing the XML Schema Data Type Representation" on page 11-3 and "Writing the Java Data Type Representation" on page 11-4; the procedure after the example lists the main steps to create such a class:

```
package examples.newTypes;

import weblogic.webservice.encoding.AbstractCodec;

import weblogic.xml.schema.binding.DeserializationContext;
import weblogic.xml.schema.binding.DeserializationException;
import weblogic.xml.schema.binding.Deserializer;
import weblogic.xml.schema.binding.SerializationContext;
import weblogic.xml.schema.binding.SerializationException;
import weblogic.xml.schema.binding.Serializer;


import weblogic.xml.stream.Attribute;
import weblogic.xml.stream.CharacterData;
import weblogic.xml.stream.ElementFactory;
import weblogic.xml.stream.EndElement;
import weblogic.xml.stream.StartElement;
import weblogic.xml.stream.XMLEvent;
import weblogic.xml.stream.XMLInputStream;
import weblogic.xml.stream.XMLName;
import weblogic.xml.stream.XMLOutputStream;
import weblogic.xml.stream.XMLStreamException;


public final class EmployeeBeanCodec extends
  weblogic.webservice.encoding.AbstractCodec
{
  public void serialize(Object obj,
                        XMLName name,
                        XMLOutputStream writer,
                        SerializationContext context)
    throws SerializationException
  {
    EmployeeBean emp = (EmployeeBean) obj;

    try {

      //outer start element
      writer.add(ElementFactory.createStartElement(name));

      //employee name element
      writer.add(ElementFactory.createStartElement("name"));
      writer.add(ElementFactory.createCharacterData(emp.getName()));
      writer.add(ElementFactory.createEndElement("name"));

      //employee id element
      writer.add(ElementFactory.createStartElement("id"));
      String id_string = Integer.toString(emp.getId());
      writer.add(ElementFactory.createCharacterData(id_string));
      writer.add(ElementFactory.createEndElement("id"));
```

```
    //outer end element
    writer.add(ElementFactory.createEndElement(name));

  } catch(XMLStreamException xse) {
    throw new SerializationException("stream error", xse);
  }
}

public Object deserialize(XMLName name,
                          XMLInputStream reader,
                          DeserializationContext context)
  throws DeserializationException
{
  // extract the desired information out of reader, consuming the
  // entire element representing the type,
  // construct your object, and return it.
  EmployeeBean employee = new EmployeeBean();

  try {
    if (reader.skip(name, XMLEvent.START_ELEMENT)) {
      StartElement top = (StartElement)reader.next();

      //next start element should be the employee name
      if (reader.skip(XMLEvent.START_ELEMENT)) {
        StartElement emp_name = (StartElement)reader.next();

        //assume that the next element is our name character data
        CharacterData cdata = (CharacterData) reader.next();
        employee.setName(cdata.getContent());
      } else {
        throw new DeserializationException("employee name not found");
      }

      //next start element should be the employee id
      if (reader.skip(XMLEvent.START_ELEMENT)) {
        StartElement emp_id = (StartElement)reader.next();

        //assume that the next element is our id character data
        CharacterData cdata = (CharacterData) reader.next();
        employee.setId(Integer.parseInt(cdata.getContent()));
      } else {
        throw new DeserializationException("employee id not found");
      }

      //we must consume our entire element to leave the stream in a
      //good state for any other deserializer
      if (reader.skip(name, XMLEvent.END_ELEMENT)) {
        XMLEvent end = reader.next();
      } else {
        throw new DeserializationException("expected end element not found");
```

```
        }
      } else {
        throw new DeserializationException("expected start element not found");
      }
    } catch (XMLStreamException xse) {
      throw new DeserializationException("stream error", xse);
    }
    return employee;
  }

  public Object deserialize(XMLName name,
                            Attribute att,
                            DeserializationContext context)
    throws DeserializationException
  {
    //NOTE: not used in this example

    // extract the desired information out of att, consuming the
    // entire element representing the type,
    // construct your object, and return it.
    return new EmployeeBean();
  }
}
```

To create the serialization class using the WebLogic XML Streaming API, follow these steps:

1. Import the following classes, which are implemented by the abstract class that your serialization class will extend:

   ```
   import weblogic.webservice.encoding.AbstractCodec;

   import weblogic.xml.schema.binding.DeserializationContext;
   import weblogic.xml.schema.binding.DeserializationException;
   import weblogic.xml.schema.binding.Deserializer;
   import weblogic.xml.schema.binding.SerializationContext;
   import weblogic.xml.schema.binding.SerializationException;
   import weblogic.xml.schema.binding.Serializer;
   ```

2. Import the WebLogic XML Streaming API classes as needed. The preceding example imports the following classes:

   ```
   import weblogic.xml.stream.Attribute;
   import weblogic.xml.stream.CharacterData;
   import weblogic.xml.stream.ElementFactory;
   import weblogic.xml.stream.EndElement;
   import weblogic.xml.stream.StartElement;
   import weblogic.xml.stream.XMLEvent;
   import weblogic.xml.stream.XMLInputStream;
   import weblogic.xml.stream.XMLName;
   ```

```
import weblogic.xml.stream.XMLOutputStream;
import weblogic.xml.stream.XMLStreamException;
```

3. Write your Java class to extend the following abstract class:

   `weblogic.webservice.encoding.AbstractCodec`

   Because JAX-RPC does not define a standard mechanism for accessing XML, the `AbstractCodec` class provides the glue to allow user written serialization classes to be used in the WebLogic Web Services runtime.

4. Implement the `serialize()` method, used to convert the data from Java to XML. The signature of this method is as follows:

   ```
   void serialize(Object obj,
                  XMLName name,
                  XMLOutputStream writer,
                  SerializationContext context)
     throws SerializationException;
   ```

   Your Java object will be contained in the `Object` parameter. Use the XML Streaming API to write the Java object to the `XMLOutputStream` parameter. Use the `XMLName` parameter as the name of the resulting element.

   Warning: Do not update the `SerializationContext` parameter; it is used internally by WebLogic Server.

5. Implement the `deserialize()` method, used to convert the data from XML to Java. The signature of this method is as follows:

   ```
   Object deserialize(XMLName name,
                      XMLInputStream reader,
                      DeserializationContext context)
     throws DeserializationException;
   ```

   The XML that you want to deserialize is contained in the `XMLInputStream` parameter. Use the WebLogic XML Streaming API to parse the XML and convert it into the returned `Object`. The `XMLName` parameter contains the expected name of the XML element.

   Call the `deserialize()` method recursively to build contained `Objects`.

   When you use the XML Streaming API to read the stream of events that make up your XML document, be sure you always finish reading an element all the way up to and including the `EndElement` event, rather than finish reading once you have read all the actual data. If you finish before reaching an `EndElement` event, the deserialization of subsequent elements might fail.

Warning: Do not update the `DeserializationContext` parameter; it is used internally by WebLogic Server.

6. If the data type for which you are creating a serialization class is used as an attribute value in your XML files, implement the following variation of the `deserialize()` method:

```
Object deserialize(XMLName name,
                   Attribute att,
                   DeserializationContext context)
  throws DeserializationException;
```

The `Attribute` parameter contains the attribute value to deserialize. The `XMLName` attribute contains the expected name of the XML element.

Warning: Do not update the `DeserializationContext` parameter; it is used internally by WebLogic Server.

Creating the Data Type Mapping File

The data type mapping file is a subset of the `web-services.xml` deployment descriptor file. It centralizes some of the information about non-built-in data types, such as the name of the Java class that describes the Java representation of the data, the name of the serialization class that converts the data between XML and Java, and so on. The `servicegen` Ant task uses this data type mapping file when creating the `web-services.xml` deployment descriptor for the WebLogic Web Service that uses the non-built-in data type.

To create the data type mapping file, follow these steps:

1. Create a text file with any name.
2. Within in the text file, add a `<type-mapping>` root element:

```
<type-mapping>
...
</type-mapping>
```

3. For each non-built-in data type for which you have created a serialization class, add a `<type-mapping-entry>` child element of the `<type-mapping>` element. Include the following attributes:
   - `xmlns:`*`name`* —Declares a namespace.
   - `class-name`—Specifies the fully qualified name of the Java class.
   - `type`—Specifies the name of XML Schema type for which this data type mapping entry applies.

  - `serializer`—The fully qualified name of the serialization class that converts the data from its Java to its XML representation. For details on creating this class, see "Writing the Serialization Class" on page 11-5.
  - `deserializer`—The fully qualified name of the serialization class that converts the data from its XML to its Java representation. For details on creating this class, see "Writing the Serialization Class" on page 11-5.

The following example shows a possible data type mapping file with one `<type-mapping>` entry for the XML Schema data type shown in "Updating the web-services.xml File With XML Schema Information" on page 11-11:

```
<type-mapping>
    <type-mapping-entry
        xmlns:p2="java:examples.newTypes"
        class-name="examples.newTypes.EmployeeBean"
        type="p2:EmployeeBean"
        serializer="examples.newTypes.EmployeeBeanCodec">
        deserializer="examples.newTypes.EmployeeBeanCodec"
    </type-mapping-entry>
</type-mapping>
```

Updating the web-services.xml File With XML Schema Information

The `web-services.xml` file generated by `servicegen` will not have the XML Schema information for the non-built-in data type for which you have created your own custom serialization class. For this reason, you must manually add the XML Schema information to the deployment descriptor, as described in the following steps:

1. In the existing `web-services.xml` file generated by the `servicegen` Ant task, find the `<types>` child element of the `<web-service>` element:

   ```
   <types>
   ...
   </types>
   ```

2. Merge your XML Schema representation of your non-built-in data type that you created in "Writing the XML Schema Data Type Representation" on page 11-3 with the any existing information within the `<types>` element, as shown in the following example:

   ```
   <types>
     <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                 xmlns:stns="java:examples.newTypes"
                 attributeFormDefault="qualified"
                 elementFormDefault="qualified"
   ```

```
                targetNamespace="java:examples.newTypes">
        <xsd:complexType name="EmployeeBean">
           <xsd:sequence>
             <xsd:element name="name"
                          type="xsd:string"
                          nillable="true"
                          minOccurs="1"
                          maxOccurs="1">
             </xsd:element>
             <xsd:element name="id"
                          type="xsd:int"
                          minOccurs="1"
                          maxOccurs="1">
             </xsd:element>
           </xsd:sequence>
        </xsd:complexType>
     </xsd:schema>
   </types>
```

CHAPTER 12

Creating SOAP Message Handlers to Intercept the SOAP Message

The following sections discuss how to use SOAP message handlers to intercept the request and response SOAP messages when developing a WebLogic Web Service:

- "Overview of SOAP Message Handlers and Handler Chains" on page 12-1
- "Creating SOAP Message Handlers: Main Steps" on page 12-2
- "Designing the SOAP Message Handlers and Handler Chains" on page 12-4
- "Implementing the Handler Interface" on page 12-6
- "Extending the GenericHandler Abstract Class" on page 12-17
- "Updating the web-services.xml File with SOAP Message Handler Information" on page 12-19
- "Using SOAP Message Handlers and Handler Chains in a Client Application" on page 12-21
- "Accessing the MessageContext of a Handler From the Backend Component" on page 12-23

Overview of SOAP Message Handlers and Handler Chains

A SOAP message handler intercepts the SOAP message in both the request and response of the Web Service. You can create handlers in both the Web Service itself and the client applications that invoke the Web Service. Refer to "Using SOAP Message Handlers to Intercept the SOAP Message" on page 4-5 for examples of when to use handlers.

The following table describes the main classes and interfaces of the `javax.xml.rpc.handler` API; later sections in this chapter describe how to use them to create handlers.

Table 12-1 JAX-RPC Handler Interfaces and Classes

javax.xml.rpc.handler Classes and Interfaces	Description
`Handler`	Main interface that you implement when creating a handler. Contains methods to handle the SOAP request, response, and faults.
`HandlerInfo`	Contains information about the handler, in particular the initialization parameters, specified in the web-services.xml file.
`MessageContext`	Abstracts the message context processed by the handler. The MessageContext properties allow the handlers in a handler chain to share processing state.
`soap.SOAPMessageContext`	Sub-interface of the MessageContext interface used to get at or update the SOAP message.
`javax.xml.soap.SOAPMessage`	Object that contains the actual request or response SOAP message, including its header, body, and attachment.

Creating SOAP Message Handlers: Main Steps

The following procedure assumes that you have already implemented and assembled a WebLogic Web Service using the `servicegen` Ant task, and you want to update the Web Service by adding handlers and handler chains.

1. Design the handlers and handler chains. See "Designing the SOAP Message Handlers and Handler Chains" on page 12-4.
2. For each handler in the handler chain, create a Java class that implements the `javax.xml.rpc.handler.Handler` interface. See "Implementing the Handler Interface" on page 12-6.

   WebLogic Server includes an extension to the JAX-RPC handler API which you can use to simplify the coding of your handler class: an abstract class called

`weblogic.webservice.GenericHandler`. See "Extending the GenericHandler Abstract Class" on page 12-17.

3. Compile the Java code into class files.
4. Update the `build.xml` file that contains the call to the `servicegen` Ant task, adding the `<handlerChain>` child element to the `<service>` element that builds your Web Service, as shown in the following example:

```
<servicegen
   destEar="ears/myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="jars/myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True" >
       <handlerChain
           name="myChain"
           handlers="myHandlers.handlerOne,
                     myHandlers.handlerTwo,
                     myHandlers.handlerThree"
       />
   </service>
</servicegen>
```

   For more information on the attributes of the `<handlerChain>` element, see "servicegen" on page B-25.

   Note: When you regenerate your Web Service using this `build.xml` file, *every* operation will be associated with the handler chain. Additionally, there is no way to specify input parameters for a handler using `servicegen`. If you want only certain operations to be associated with this handler chain, or you prefer not to regenerate your Web Service using `servicegen`, you can update the `web-services.xml` file of your WebLogic Web Service manually. For details, see "Updating the web-services.xml File with SOAP Message Handler Information" on page 12-19.

5. Re-run the `servicegen` Ant task to regenerate your Web Service.

For information about creating client-side SOAP message handlers and handler chains, see "Using SOAP Message Handlers and Handler Chains in a Client Application" on page 12-21.

Designing the SOAP Message Handlers and Handler Chains

When designing your SOAP message handlers, you must decide:

- The number of handlers needed to perform all the work
- The sequence of execution
- Whether to invoke a back-end component or whether the Web Service consists of only a handler chain.

Each handler in a handler chain has one method for handling the request SOAP message and another method for handling the response SOAP message. You specify the handlers in the `web-services.xml` deployment descriptor file. An ordered group of handlers is referred to as a *handler chain*.

When invoking a Web Service, WebLogic Server executes handlers as follows:

1. The `handleRequest()` methods of the handlers in the handler chain are all executed, in the order specified in the `web-services.xml` file. Any of these `handleRequest()` methods might change the SOAP message request.
2. When the `handleRequest()` method of the last handler in the handler chain executes, WebLogic Server invokes the back-end component that implements the Web Service, passing it the final SOAP message request.

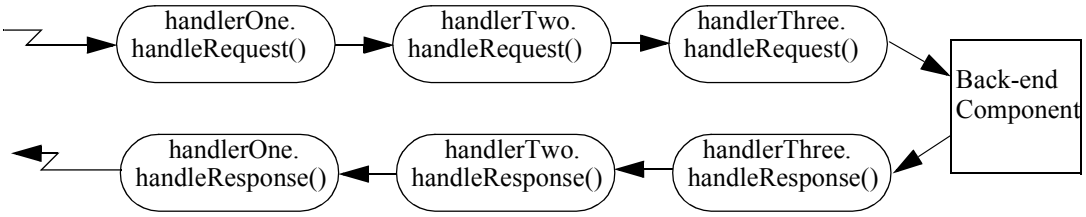
   Note: This step only occurs if a back-end component has actually been defined for the Web Service; it is possible to develop a Web Service that consists of only a handler chain.
3. When the back-end component has finished executing, the `handleResponse()` methods of the handlers in the handler chain are executed in the *reverse* order specified in the `web-services.xml` file. Any of these `handleResponse()` methods might change the SOAP message response.
4. When the `handleResponse()` method of the first handler in the handler chain executes, WebLogic server returns the final SOAP message response to the client application that invoked the Web Service.

For example, assume that you have specified a handler chain called `myChain` that contains three handlers in the `web-services.xml` deployment descriptor, as shown in the following excerpt:

```
<handler-chains>
   <handler-chain name="myChain">
     <handler class-name="myHandlers.handlerOne" />
     <handler class-name="myHandlers.handlerTwo" />
     <handler class-name="myHandlers.handlerThree" />
   </handler-chain>
</handler-chains>
```

The following graphic shows the order in which WebLogic Server executes the `handleRequest()` and `handleResponse()` methods of each handler:

Figure 12-1 Order of Execution of Handler Methods



Each SOAP message handler has a separate method to process the request and response SOAP message because the same type of processing typically must happen in both places. For example, you might design an Encryption handler whose `handleRequest()` method decrypts secure data in the SOAP request and `handleResponse()` method encrypts the SOAP response.

You can, however, design a handler that process only the SOAP request and does no equivalent processing of the response.

You can also choose not to invoke the next handler in the handler chain and send an immediate response to the client application at any point. The way to do this is discussed in later sections.

Finally, you can design a Web Service that contains only handlers in a handler chain, and no back-end component at all. In this case, when the `handleRequest()` method in the last handler has executed, the chain of `handleResponse()` methods is automatically invoked. See "Updating the web-services.xml File with SOAP Message Handler Information" on page 12-19 for an example of using the `web-services.xml` file to specify that only a handler chain, and no back-end component, implements a Web Service.

Implementing the Handler Interface

Your SOAP message handler class must implement the `javax.rpc.xml.handler.Handler` interface, as shown in the example at the end of this section. In particualr, the `Handler` interface contains the following methods that you must implement:

- `init()`

  See "Implementing the Handler.init() Method" on page 12-8.

- `destroy()`

  See "Implementing the Handler.destroy() Method" on page 12-8.

- `getHeaders()`

  See "Implementing the Handler.getHeaders() Method" on page 12-9.

- `handleRequest()`

  See "Implementing the Handler.handleRequest() Method" on page 12-9.

- `handleResponse()`

  See "Implementing the Handler.handleResponse() Method" on page 12-10.

- `handleFault()`

  See "Implementing the Handler.handleFault() Method" on page 12-11.

Sometimes you might need to directly view or update the SOAP message from within your handler, in particular when handling attachments, such as image. In this case, use the `javax.xml.soap.SOAPMessage` abstract class, which is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification For details, see "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 12-12.

The following example demonstrates a simple SOAP message handler that prints out the SOAP request and response messages:

```
package examples.webservices.handler.log;

import java.util.Map;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;

import weblogic.logging.NonCatalogLogger;
```

```
/**
 * Class that implements a handler in the handler chain, used to access the SOAP
 * request and response message.
 * <p>
 * The class implements the <code>javax.xml.rpc.handler.Handler</code>
 *  interface. The class simply prints the SOAP request and response messages
 * to a log file before the messages are processed by the backend component.
 *
 * @author Copyright (c) 2003 by BEA Systems. All Rights Reserved.
 */

public final class LogHandler
  implements Handler
{
  private NonCatalogLogger log;

  private HandlerInfo handlerInfo;

  /**
   *  Initializes the instance of the handler.  Creates a nonCatalogLogger to
   *  log messages to.
   */
  public void init(HandlerInfo hi) {
    log = new NonCatalogLogger("WebService-LogHandler");
    handlerInfo = hi;
  }

  /**
   * Destroys the Handler instance.
   */
  public void destroy() {}

  public QName[] getHeaders() { return handlerInfo.getHeaders(); }

  /**
   * Specifies that the SOAP request message be logged to a log file before the
   * message is sent to the Java class backend component
   */
  public boolean handleRequest(MessageContext mc) {
    SOAPMessageContext messageContext = (SOAPMessageContext) mc;

    System.out.println("** Request: "+messageContext.getMessage().toString());
    log.info(messageContext.getMessage().toString());
    return true;
  }

  /**
   * Specifies that the SOAP response message be logged to a log file before the
   * message is sent back to the client application that invoked the Web service.
```

```
 */
public boolean handleResponse(MessageContext mc) {

  SOAPMessageContext messageContext = (SOAPMessageContext) mc;

  System.out.println("** Response: "+messageContext.getMessage().toString());
  log.info(messageContext.getMessage().toString());
  return true;
}

/**
 * Specifies that a message be logged to the log file if a SOAP fault is
 * thrown by the Handler instance.
 */
public boolean handleFault(MessageContext mc) {
  SOAPMessageContext messageContext = (SOAPMessageContext) mc;

  System.out.println("** Fault: "+messageContext.getMessage().toString());
  log.info(messageContext.getMessage().toString());
      return true;
}
```

Implementing the Handler.init() Method

The `Handler.init()` method is called to create an instance of a `Handler` object and to enable the instance to initialize itself. Its signature is:

```
public void init(HandlerInfo config) throws JAXRPCException {}
```

The `HandlerInfo` object contains information about the SOAP message handler, in particular the initialization parameters, specified in the `web-services.xml` file. Use the `HandlerInfo.getHandlerConfig()` method to get the parameters; the method returns a `Map` object that contains name-value pairs.

Implement the `init()` method if you need to process the initialization parameters or if you have other initialization tasks to perform.

Sample uses of initialization parameters are to turn debugging on or off, specify the name of a log file to which to write messages or errors, and so on.

Implementing the Handler.destroy() Method

The `Handler.destroy()` method is called to destroy an instance of a `Handler` object. Its signature is:

```
public void destroy() throws JAXRPCException {}
```

Implement the `destroy()` method to release any resources acquired throughout the handler's lifecycle.

Implementing the Handler.getHeaders() Method

The `Handler.getHeaders()` method gets the header blocks processed by this Handler instance. Its signature is:

```
public QName[] getHeaders() {}
```

Implementing the Handler.handleRequest() Method

The `Handler.handleRequest()` method is called to intercept a SOAP message request before it is processed by the back-end component. Its signature is:

```
public boolean handleRequest(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to decrypt data in the SOAP message before it is processed by the back-end component, to make sure that the request contains the correct number of parameters, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message request. The SOAP message request itself is stored in a `javax.xml.soap.SOAPMessage` object. For detailed information on this object, see "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 12-12.

The `SOAPMessageContext` class defines two methods for processing the SOAP request:

- `SOAPMessageContext.getMessage()`returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message request.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`updates the SOAP message request after you have made changes to it.

After you code all the processing of the SOAP request, do one of the following:

- Invoke the next handler on the handler request chain by returning `true`.

  The next handler on the request chain is specified as the next `<handler>` subelement of the `<handler-chain>` element in the `web-services.xml` deployment descriptor. If there are no more handlers in the chain, the method either invokes the back-end component,

passing it the final SOAP message request, or invokes the `handleResponse()` method of the last handler, depending on how you have configured your Web Service.

- Block processing of the handler request chain by returning `false`.

  Blocking the handler request chain processing implies that the back-end component does not get executed for this invoke of the Web Service. You might want to do this if you have cached the results of certain invokes of the Web Service, and the current invoke is on the list.

  Although the handler request chain does not continue processing, WebLogic Server does invoke the handler *response* chain, starting at the current handler. For example, assume that a handler chain consists of two handlers: handlerA and handlerB, where the `handleRequest()` method of handlerA is invoked before that of handlerB. If processing is blocked in handlerA (and thus the `handleRequest()` method of handlerB is *not* invoked), the handler response chain starts at handlerA and the `handleRequest()` method of handlerB is not invoked either.

- Throw the `javax.xml.rpc.soap.SOAPFaultException` to indicate a SOAP fault.

  If the `handleRequest()` method throws a `SOAPFaultException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, and invokes the `handleFault()` method of this handler.

- Throw a `JAXRPCException` for any handler specific runtime errors.

  If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the `handleFault()` method of this handler.

Implementing the Handler.handleResponse() Method

The `Handler.handleResponse()` method is called to intercept a SOAP message response after it has been processed by the back-end component, but before it is sent back to the client application that invoked the Web Service. Its signature is:

```
public boolean handleResponse(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to encrypt data in the SOAP message before it is sent back to the client application, to further process returned values, and so on.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message response. The SOAP message response itself is stored in a `javax.xml.soap.SOAPMessage` object. See "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 12-12.

The `SOAPMessageContext` class defines two methods for processing the SOAP response:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message response.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message response after you have made changes to it.

After you code all the processing of the SOAP response, do one of the following:

- Invoke the next handler on the handler response chain by returning `true`.

  The next response on the handler chain is specified as the preceding `<handler>` subelement of the `<handler-chain>` element in the `web-services.xml` deployment descriptor. (Remember that responses on the handler chain execute in the *reverse* order that they are specified in the `web-services.xml` file. See "Designing the SOAP Message Handlers and Handler Chains" on page 12-4 for more information.)

  If there are no more handlers in the chain, the method sends the final SOAP message response to the client application that invoked the Web Service.

- Block processing of the handler response chain by returning `false`.

  Blocking the handler response chain processing implies that the remaining handlers on the response chain do not get executed for this invoke of the Web Service and the current SOAP message is sent back to the client application.

- Throw a `JAXRPCException` for any handler specific runtime errors.

  If the `handleRequest()` method throws a `JAXRPCException`, WebLogic Server catches the exception, terminates further processing of the handler request chain, logs the exception to the WebLogic Server logfile, and invokes the `handleFault()` method of this handler.

Implementing the Handler.handleFault() Method

The `Handler.handleFault()` method processes the SOAP faults based on the SOAP message processing model. Its signature is:

```
public boolean handleFault(MessageContext mc) throws JAXRPCException  {}
```

Implement this method to handle processing of any SOAP faults generated by the `handleResponse()` and `handleRequest()` methods, as well as faults generated by the back-end component.

The `MessageContext` object abstracts the message context processed by the SOAP message handler. The `MessageContext` properties allow the handlers in a handler chain to share processing state.

Use the `SOAPMessageContext` sub-interface of `MessageContext` to get at or update the contents of the SOAP message. The SOAP message itself is stored in a `javax.xml.soap.SOAPMessage` object. See "Directly Manipulating the SOAP Request and Response Message Using SAAJ" on page 12-12.

The `SOAPMessageContext` class defines the following two methods for processing the SOAP message:

- `SOAPMessageContext.getMessage()`: returns a `javax.xml.soap.SOAPMessage` object that contains the SOAP message.
- `SOAPMessageContext.setMessage(javax.xml.soap.SOAPMessage)`: updates the SOAP message after you have made changes to it.

After you code all the processing of the SOAP fault, do one of the following:

- Invoke the `handleFault()` method on the next handler in the handler chain by returning `true`.
- Block processing of the handler fault chain by returning `false`.

Directly Manipulating the SOAP Request and Response Message Using SAAJ

The `javax.xml.soap.SOAPMessage` abstract class is part of the SOAP With Attachments API for Java 1.1 (SAAJ) specification. You use the class to manipulate request and response SOAP messages when creating SOAP message handlers. This section describes the basic structure of a `SOAPMessage` object and some of the methods you can use to view and update a SOAP message.

A `SOAPMessage` object consists of a `SOAPPart` object (which contains the actual SOAP XML document) and zero or more attachments.

Refer to the SAAJ Javadocs for the full description of the `SOAPMessage` class. For more information on SAAJ, go to http://java.sun.com/xml/saaj/index.html.

The SOAPPart Object

The `SOAPPart` object contains the XML SOAP document inside of a `SOAPEnvelope` object. You use this object to get the actual SOAP headers and body.

The following sample Java code shows how to retrieve the SOAP message from a `MessageContext` object, provided by the `Handler` class, and get at its parts:

```
SOAPMessage soapMessage =  messageContext.getRequest();
SOAPPart soapPart = soapMessage.getSOAPPart();
SOAPEnvelope soapEnvelope = soapPart.getEnvelope();
SOAPBody soapBody = soapEnvelope.getBody();
SOAPHeader soapHeader = soapEnvelope.getHeader();
```

The AttachmentPart Object

The `javax.xml.soap.AttachmentPart` object contains the optional attachments to the SOAP message. Unlike the rest of a SOAP message, an attachment is not required to be in XML format and can therefore be anything from simple text to an image file.

Warning: If you are going to access a `java.awt.Image` attachment from your SOAP message handler, see "Manipulating Image Attachments in a SOAP Message Handler" on page 12-15 for important information.

Use the following methods of the `SOAPMessage` class to manipulate the attachments:

- `countAttachments()`: returns the number of attachments in this SOAP message.
- `getAttachments()`: retrieves all the attachments (as `AttachmentPart` objects) into an `Iterator` object.
- `createAttachmentPart()`: create an `AttachmentPart` object from another type of `Object`.
- `addAttachmentPart()`: adds an `AttachmentPart` object, after it has been created, to the `SOAPMessage`.

The following example shows how you can create a SOAP message handler that accesses the SOAP attachment using the SAAJ API. The example uses the `weblogic.webservice.GenericHandler` abstract class, which is a WebLogic Server extension to the JAX-RPC handler API. For details about the `GenericHandler` class, see "Extending the GenericHandler Abstract Class" on page 12-17.

```
import java.util.Iterator;
```

```
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.JAXRPCException;

import javax.xml.soap.AttachmentPart;
import javax.xml.soap.SOAPException;
import javax.xml.soap.SOAPMessage;

import weblogic.webservice.GenericHandler;

import weblogic.utils.Debug;

public final class ServerHandler
  extends GenericHandler
{
  public boolean handleRequest(MessageContext m) {

    SOAPMessageContext ctx = (SOAPMessageContext) m;

    SOAPMessage request = ctx.getMessage();

    if (request.countAttachments() == 0) {
      throw new JAXRPCException("** Expected attachments");
    }

    Iterator it = request.getAttachments();

    try {
      while(it.hasNext()) {
        AttachmentPart part = (AttachmentPart) it.next();
        Debug.say("** Received attachment: "+
          part.getContent());
      }
    } catch (SOAPException e) {
      e.printStackTrace();
      throw new JAXRPCException(e);
    }

    return true;

  }

public boolean handleResponse(MessageContext m) {

    SOAPMessageContext ctx = (SOAPMessageContext) m;

    SOAPMessage response = ctx.getMessage();

    if (response.countAttachments() != 0) {
      throw new JAXRPCException("** Expected no attachments");
    }
```

```
    AttachmentPart part = response.createAttachmentPart();

    part.setContent("<weblogic>RESPONSE</weblogic>", "text/xml");
    response.addAttachmentPart(part);

    return true;
  }
}
```

Manipulating Image Attachments in a SOAP Message Handler

It is assumed in this section that you are creating a SOAP message handler that accesses a `java.awt.Image` attachment and that the `Image` has been sent from a client application that uses the client JAX-RPC stubs generated by the `clientgen` Ant task.

In the client code generated by the `clientgen` Ant task, a `java.awt.Image` attachment is sent to the invoked WebLogic Web Service with a MIME type of `text/xml` rather than `image/gif`, and the Image is serialized into a stream of integers that represents the image. In particular, the client code serializes the `Image` using the following format:

- `int width`
- `int height`
- `int[] pixels`

This means that, in your SOAP message handler that manipulates the received Image attachment, you must deserialize this stream of data to then re-create the original Image.

The following example shows an implementation of the `handleRequest()` method of the `Handler` interface that iterates through the attachments of a SOAP message, and for each attachment, gets the input stream, deserializes it into a `java.awt.Image`, and then displays it in a frame using the Java Swing classes. It is assumed in the handler that all attachments are Images.

```
// it is assumed in this handler that all attachments are Image attachments

  public boolean handleRequest(MessageContext mc)
  {
     try {
       SOAPMessageContext messageContext = (SOAPMessageContext) mc;

       SOAPMessage soapmsg = messageContext.getMessage();
       Iterator iter = soapmsg.getAttachments();

       // iterate through the attachments
       while ( iter.hasNext() ) {
         AttachmentPart part = (AttachmentPart) iter.next();
```

```
        // get the input stream from the attachment and read the bytes into a
byte[]

        DataHandler dh = part.getDataHandler();
        InputStream is = dh.getInputStream();
        int size = is.available();
        byte[] bytes = new byte[size];
        is.read( bytes, 0, size);

        // create a String from the byte[]
        String content = new String(bytes);

        // decode the String
        byte[] bin =
weblogic.xml.schema.types.XSDBase64Binary.convertXml(content );

        // get an input stream for the binary object
        ByteArrayInputStream in = new ByteArrayInputStream( bin );
        ObjectInputStream oin = new ObjectInputStream( in );

        // deserialize the stream.
        // the format for an image is:
        // int width
        // int height
        // int[] pix -- an array of pixels
        int width = oin.readInt();
        int height = oin.readInt();
        int[] pix = (int[])oin.readObject();

        // create an Image from the deserialized pieces
        java.awt.image.MemoryImageSource source =
          new java.awt.image.MemoryImageSource(width, height, pix, 0, width);

        // this is sample code for displaying the image in a frame
        java.awt.Panel panel = new java.awt.Panel();
        java.awt.Image image = panel.createImage( source );

       javax.swing.ImageIcon ii = new javax.swing.ImageIcon(image);
       javax.swing.JLabel label = new javax.swing.JLabel(ii);
       javax.swing.JFrame mainframe = new javax.swing.JFrame();
       mainframe.getContentPane().add(label);
       mainframe.pack();
       mainframe.setVisible(true);
     }
   } catch (Exception ex) { ex.printStackTrace(); }

   return true;
 }
```

Extending the GenericHandler Abstract Class

WebLogic Server includes an extension to the JAX-RPC handler API that you can use to simplify the Java code of your SOAP message handler class. This extension is the abstract class `weblogic.webservices.GenericHandler`. It implements the JAX-RPC `javax.xml.rpc.handler.Handler` interface.

Note: The `weblogic.webservices.GenericHandler` abstract class was originally developed for WebLogic Server when the JAX-RPC specification was not yet final and did not include this functionality. However, now that JAX-RPC includes its own `GenericHandler` class which is almost exactly the same as the WebLogic Server class, BEA highly recommends that you use the standard JAX-RPC abstract class rather than the WebLogic-specific one. The documentation in this section is provided for compatibility reasons only. For more information about the JAX-RPC `javax.xml.rpc.handler.GenericHandler` abstract class, see the JAX-RPC Javadocs.

Because `GenericHandler` is an abstract class, you need only implement the methods that will contain actual code, rather than having to implement every method of the `Handler` interface even if the method does nothing. For example, if your handler does not use initialization parameters and you do not need to allocate any additional resources, you do not need to implement the `init()` method.

The GenericHandler class is defined as follows:

```
package weblogic.webservice;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */

public abstract class GenericHandler
  implements Handler
{

  private HandlerInfo handlerInfo;

  public void init(HandlerInfo handlerInfo) {
    this.handlerInfo = handlerInfo;
  }

  protected HandlerInfo getHandlerInfo() { return handlerInfo; }
```

```
  public boolean handleRequest(MessageContext msg) {
    return true;
  }

  public boolean handleResponse(MessageContext msg) {
    return true;
  }

  public boolean handleFault(MessageContext msg) {}

  public void destroy() {}
  public QName[] getHeaders() { return handlerInfo.getHeaders(); }

}
```

The following sample code, taken from the `examples.webservices.handler.nocomponent` product example, shows how to use the `GenericHandler` abstract class to create your own handler. The example implements only the `handleRequest()` and `handleResponse()` methods. It does not implement (and thus does not include in the code) the `init()`, `destroy()`, `getHeaders()`, and `handleFault()` methods.

```
package examples.webservices.handler.nocomponent;

import java.util.Map;

import javax.xml.rpc.JAXRPCException;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.soap.*;

import weblogic.webservice.GenericHandler;

import weblogic.utils.Debug;

/**
 * @author Copyright (c) 2002 by BEA Systems. All Rights Reserved.
 */

public final class EchoStringHandler
  extends GenericHandler
{
  private int me = System.identityHashCode(this);

  public boolean handleRequest(MessageContext messageContext) {
    System.err.println("** handleRequest called in: "+me);
    return true;
  }

  public boolean handleResponse(MessageContext messageContext) {
```

```
    try {
      MessageFactory messageFactory = MessageFactory.newInstance();

      SOAPMessage m = messageFactory.createMessage();

      SOAPEnvelope env = m.getSOAPPart().getEnvelope();

      SOAPBody body = env.getBody();

      SOAPElement fResponse =
        body.addBodyElement(env.createName("echoResponse"));

      fResponse.addAttribute(env.createName("encodingStyle"),
        "http://schemas.xmlsoap.org/soap/encoding/");

      SOAPElement result =
        fResponse.addChildElement(env.createName("result"));

      result.addTextNode("Hello World");

      ((SOAPMessageContext)messageContext).setMessage(m);

      return true;

    } catch (SOAPException e) {
      e.printStackTrace();
      throw new JAXRPCException(e);
    }
  }
}
```

Updating the web-services.xml File with SOAP Message Handler Information

The `web-services.xml` deployment descriptor file describes the SOAP message handlers and handler chains defined for a Web Service and the order in which they should be executed.

To update the `web-services.xml` file with handler information:

1. Create a `<handler-chains>` child element of the `<web-services>` root element that will contain a list of all handler chains defined for the Web Service.
2. Create a `<handler-chain>` child element of the `<handler-chains>` element; within this element list all the handlers in the handler chain. For each handler, use the `class-name` attribute to specify the fully qualified name of the Java class that implements the handler. Use the `<init-params>` element to specify any initialization parameters of the handler.

The following sample excerpt shows a handler chain called `myChain` that contains three handlers, the first of which has an initialization parameter:

```
<web-services>
   <handler-chains>
      <handler-chain name="myChain">
        <handler class-name="myHandlers.handlerOne" >
            <init-params>
               <init-param name="debug" value="on" />
            </init-params>
        </handler>
        <handler class-name="myHandlers.handlerTwo" />
        <handler class-name="myHandlers.handlerThree" />
      </handler-chain>
   </handler-chains>
...
</web-services>
```

3. Use the `<operation>` child element of the `<operations>` element (which itself is a child of the `<web-service>` element) to specify that the handler chain is an operation of the Web Service. Follow one of the next two scenarios:

   – The handler chain executes together with a back-end component, such as a stateless session EJB.

     In this case use the `component`, `method`, and `handler-chain` attributes of the `<operation>` element, as shown in the following partial excerpt of a `web-services.xml` file:

     ```
     <web-service>
       <components>
         <stateless-ejb name="myEJB">
           ...
         </stateless-ejb>
       </components>
       <operations>
         <operation name="getQuote"
                    method="getQuote"
                    component="myEJB"
                    handler-chain="myChain" />
       </operations>
     </web-service>
     ```

     In the example, the request chain of the `myChain` handler chain executes first, then the `getQuote()` method of the `myEJB` stateless session EJB component, and finally the response chain of `myChain`.

   – The handler chain executes on its own, *without* a back-end component.

In this case use only the handler-chain attribute of the `<operation>` element and explicitly do not specify the component or method attributes, as shown in the following excerpt:

```
<web-service>
  <operations>
    <operation name="chainService"
               handler-chain="myChain" />
   </operations>
</web-service>
```

In the example, the Web Service consists solely of the `myChain` handler chain.

Using SOAP Message Handlers and Handler Chains in a Client Application

Most of this chapter describes how to create SOAP message handlers in a handler chain that execute as part of the Web Service running on WebLogic Server. You can also create handlers that execute in a client application. In the case of a client-side handler, the handler executes twice when a client application invokes a Web Service:

- directly before the client application sends the SOAP request to the Web Service
- directly after the client application receives the SOAP response from the Web Service

You create a client-side handler in the same way you create a server-side handler: write a Java class that implements the `javax.rpc.xml.handler.Handler` interface. In many cases you can use the exact same handler class on both the Web Service running on WebLogic Server *and* the client applications that invoke the Web Service. For example, you can write a generic logging handler class that logs all sent and received SOAP messages, both for the server and for the client. For details about writing the handler Java class, see "Implementing the Handler Interface" on page 12-6.

After you have created your client-side handler class, the process for registering the handler on the client application is different from that of the server. Because client applications do not have deployment descriptors, you must register the handler programmatically using the `javax.xml.rpc.handler.HandlerInfo` and `javax.xml.rpc.handler.HandlerRegistry` classes. The following sample client application shows how to do this, with relevant sections in bold discussed after the example:

```
import java.util.ArrayList;

import java.io.IOException;

import javax.xml.namespace.QName;

import javax.xml.rpc.ServiceException;

import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;

public class Main{

  public static void main( String[] args ){

    if( args.length == 1 ){
      new Main( args[0] );
    }else{
      throw new IllegalArgumentException( "URL of the service not specified" );
    }
  }

  public Main( String wsdlUrl ){
    try{

      HelloWorldService service = new HelloWorldService_Impl( wsdlUrl );
      HelloWorldServicePort port = service.getHelloWorldServicePort();

      QName portName = new QName( "http://tutorial/sample4/",
        "HelloWorldServicePort");

      HandlerRegistry registry = service.getHandlerRegistry();

      List handlerList = new ArrayList();
      handlerList.add( new HandlerInfo( ClientHandler.class, null, null ) );

      registry.setHandlerChain( portName, handlerList );

      System.out.println( port.helloWorld() );
    }catch( IOException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( ServiceException e ){
      System.out.println( "Failed to create web service client:" + e );
    }
  }
}
```

The main points to notice about the example are as follows:

- Import the JAX-RPC `HandlerInfo` and `HandlerRegistry` classes which will be used to register the client-side handler class:

```
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;
```

- Create a `QName` object that contains the qualified name of the Web Service port:

  ```
  QName portName = new QName( "http://tutorial/sample4/",
         "HelloWorldServicePort");
  ```

  Refer to the WSDL of the Web Service you are invoking for the name of the port and its namespace.

- Create a `HandlerRegistry` object:

  ```
  HandlerRegistry registry = service.getHandlerRegistry();
  ```

- Create a `List` object that contains a list of the handlers you want to register. This list becomes the client-side handler chain. Use the `HandlerInfo` class to specify the name of your Java handler class:

  ```
  List handlerList = new ArrayList();
  handlerList.add( new HandlerInfo( ClientHandler.class, null, null ) );
  ```

  In the example, the handler chain consists of just one handler: `ClientHandler.class`. You can, however, create a handler chain of as many handlers as you want.

  Warning: The order in which you add the handlers to the `List` object specifies the order in which the handlers are executed in the client application. For example, if you want `HandlerA.class` to execute first and then `HandlerB.class`, be sure you add `HandlerA.class` to the list before `HandlerB.class`.

- Register the handler chain with the client application using the `HandlerRegistry.setHandlerChain()` method:

  ```
  registry.setHandlerChain( portName, handlerList );
  ```

Accessing the MessageContext of a Handler From the Backend Component

When you create a handler by implementing the `Handler` interface, you can set properties of the `MessageContext` object in one of the handler methods (such as `handleRequest()`), by using the `MessageContext.setProperty()` method. To access these properties from the backend component that is invoked after the handler chain, you must use the `weblogic.webservice.context.WebServiceContext` API to get the `MessageContext`.

For example, the following code snippet shows an implementation of the `Handler.handleRequest()` method in which a user-defined property `TransID` is set for the `MessageContext` object:

```
import javax.xml.rpc.handler.MessageContext;

 ...

 public boolean handleRequest(MessageContext mc) {
   try {
      mc.setProperty("TransId", "AX123");
   }
   catch (Exception ex) {
     System.out.println("exception from Handler: " + ex.getLocalizedMessage());
   }

   return true;
}
```

The following sample code from the Java class that implements the backend component shows how to access the `TransID` property using the `weblogic.webservice.context.WebServiceContext` API:

```
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.rpc.handler.MessageContext;
import weblogic.webservice.context.WebServiceContext;

 ...

  public String someMethod(String s) {
   try {
      SOAPMessageContext soapMessageContext =
         WebServiceContext.currentContext().getLastMessageContext();
      String somePropery = (String)soapMessageContext.getProperty("TransId");
      System.out.println("TransId =" + someProperty);

   }
   catch(Exception ex) {
     System.out.println("exception from service: " + ex.getLocalizedMessage());
   }

   return s;
  }
```

CHAPTER 13

Configuring Security

The following sections describe how to configure security for WebLogic Web Services:

- "Overview of Web Services Security" on page 13-1
- "What Type of Security Should You Configure?" on page 13-2
- "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 13-3
- "Configuring Transport-Level Security (SSL): Main Steps" on page 13-31
- "Configuring SSL for a Client Application" on page 13-32
- "Configuring Access Control Security: Main Steps" on page 13-41
- "Testing a Secure WebLogic Web Service From Its Home Page" on page 13-46

Overview of Web Services Security

To secure your WebLogic Web Service, you configure one or more of three conceptually different types of security:

- Message-level security, in which data in a SOAP message is digitally signed or encrypted.

  See "Configuring Message-Level Security (Digital Signatures and Encryption)" on page 13-3.

- Transport-level security, in which SSL is used to secure the connection between a client application and the Web Service.

  See "Configuring Transport-Level Security (SSL): Main Steps" on page 13-31.

- Access control security, which specifies which users, groups, and roles are allowed to access Web Services.

  See "Configuring Access Control Security: Main Steps" on page 13-41.

What Type of Security Should You Configure?

Access control security answers the question "who can do what?" First you specify the list of users, groups, or roles that are allowed to access a Web Service (or the component that implement the Web Service). Then, when a client application attempts to invoke a Web Service operation, the client authenticates itself to WebLogic Server, using HTTP, and if the client has the authorization, it is allowed to continue with the invocation. Access control security secures only WebLogic Server resources. This means that if you configure *only* access control security, the connection between the client application and WebLogic Server is not secure and the SOAP message is in plain text.

With transport-level security, you secure the connection between the client application and WebLogic Server with Secure Sockets Layer (SSL). SSL provides secure connections by allowing two applications connecting over a network to authenticate the other's identity and by encrypting the data exchanged between the applications. Authentication allows a server, and optionally a client, to verify the identity of the application on the other end of a network connection. Encryption makes data transmitted over the network intelligible only to the intended recipient.

Transport-level security, however, secures only the connection itself. This means that if there is an intermediary between the client and WebLogic Server, such as a router or message queue, the intermediary gets the SOAP message in plain text. When the intermediary sends the message to a second receiver, the second receiver does not know who the original sender was. Additionally, the encryption used by SSL is "all or nothing": either the entire SOAP message is encrypted or it is not encrypted at all. There is no way to specify that only selected parts of the SOAP message be encrypted.

Message-level security includes all the security benefits of SSL, but with additional flexibility and features. Message-level security is end-to-end, which means that a SOAP message is secure even when the transmission involves one or more intermediaries. The SOAP message itself is digitally signed and encrypted, rather than just the connection. And finally, you can specify that only parts of the message be signed or encrypted.

Configuring Message-Level Security (Digital Signatures and Encryption)

Message-level security specifies whether the SOAP messages between a client application and the Web Service it is invoking should be digitally signed or encrypted or both.

WebLogic Web Services implement the following OASIS Standard 1.0 Web Services Security specifications, dated April 6 2004:

- Web Services Security: SOAP Message Security
- Web Services Security: Username Token Profile
- Web Services Security: X.509 Token Profile

These specifications provide three main mechanisms: security token propagation, message integrity, and message confidentiality. These mechanisms can be used independently (such as passing a username security token for user authentication) or together (such as digitally signing and encrypting a SOAP message.)

The following sections provide information about message-level security:

- "Main Use Cases" on page 13-3
- "Unimplemented Features of the Web Services Security Core Specification" on page 13-4
- "Terminology" on page 13-5
- "Architectural Overview of Message-Level Security" on page 13-5
- "Configuring Message-Level Security: Main Steps" on page 13-9

Main Use Cases

BEA's implemenation of the *Web Services Security: SOAP Message Security* specification is designed to fully support the following use cases:

- Use an X.509 certificate to encrypt and sign a SOAP message, starting from the client application that invokes the message-secured Web Service, to the WebLogic Server instance that is hosting the Web Service and back to the client application. The SOAP message itself contains all the security information, so intermediaries between the client application and Web Service can also play a part without compromising any security of the message.

- Provide flexibility over what parts of the SOAP message are signed and encrypted. By default, when you enable WebLogic Web Service message-level security, the entire SOAP message body is encrypted and signed. You can, however, specify that all occurrences of a specific element in the SOAP message be signed, encrypted, or both.
- Include an encrypted and signed username and password in the SOAP message (rather than in the HTTP header, as is true for SSL and access control security) for further downstream processing.

Unimplemented Features of the Web Services Security Core Specification

WebLogic Web Services do not implement all features of the Web Services Security Core specification as follows:

- The specification allows for any type of security token to be passed in the SOAP message. WebLogic Web Services, however, supports only two types of tokens:
  - `UsernameToken`
  - `BinarySecurityToken`

    Additionally, the only supported value for the `ValueType` attribute of the `BinarySecurityToken` element is `wsse:X509v3`.

  WebLogic Web Services do not support (among others) custom, SAML, Kerberos, and XrML tokens.
- WebLogic Web Services uses the default Identity Asserter to map certificates to valid users. The default Identity Asserter, however, does not verify that a client application's digital certificate used to sign or encrypt a SOAP message was issued by a trusted certificate authority (CA). If you require this type of validation, you must write a custom Identity Assertion provider for X.509 that validates the digital certificate as part of the mapping of a user to a Subject.

  For more information, see the Identity Assertion Providers section of the Developing Security Providers for WebLogic Server guide.
- WebLogic Web Services do not support the XML Decryption Transformation algorithm when signing a SOAP message. WebLogic Web Services support only the Exclusive XML Canonicalization algorithm.
- A message-secured WebLogic Web Service first signs and then encrypts the out-going SOAP response. You cannot change this order.

- WebLogic Web Services do not support secret key encryption; they support only public key encryption.

Terminology

Note the following terms:

- *key pair*: pair of public and private keys.
- *digital certificate*: binding of key pairs to an identity, such as a username.
- *keystore*: file that stores key pairs and digital certificates securely.

Architectural Overview of Message-Level Security

The `<security>` element of the `web-services.xml` deployment descriptor file specifies whether a WebLogic Web Service has been configured for message-level security. In particular, the `<security>` element describes:

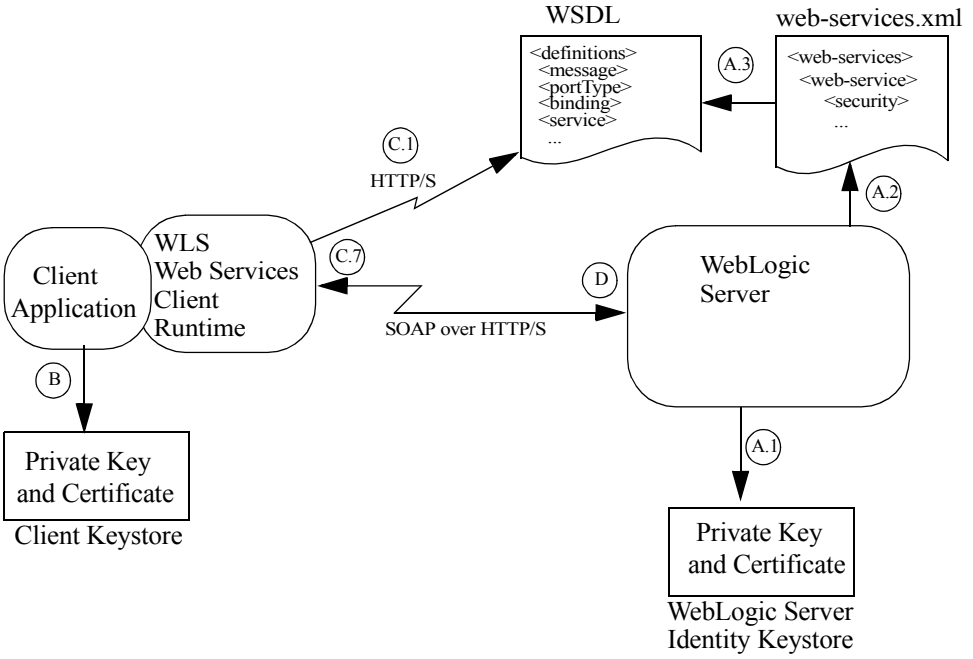
- For a particular message-secured operation, what parts of the SOAP request and response should be encrypted or digitally signed.
- The encryption and signature key pairs that WebLogic Server uses from its identity keystore to sign, verify, encrypt, and decrypt the SOAP messages.
- Which operations have been configured for message-level security.

When the Web Service is deployed, the security information specified in the `web-services.xml` file is published in the WSDL so that client applications that invoke the Web Service know whether they need to digitally sign or encrypt the SOAP messages.

Note: Because WSDL 1.1 does not include a standard for specifying security information, the way that WebLogic Server publishes its message-level security information is proprietary.

The following diagram and paragraphs describe what happens when a message-secured WebLogic Web Service is deployed and a client application invokes it. The paragraphs are broken up according to the actor that performs the action.

Figure 13-1 Message-Secured WebLogic Web Service Architecture



A. WebLogic Server

1. Loads the key pairs and certificates from WebLogic Server's identity keystore.
2. Deploys the message-secured Web Service, using information from the `web-services.xml` deployment descriptor, such as which operations require what type of message-level security.
3. Updates the WSDL of the Web service with security information so that client applications that invoke the Web Service know what security processing needs to occur on the SOAP request. This security information includes the certificate for WebLogic Server's encryption key pair, used by the client application to encrypt the SOAP request. WebLogic Server uses the information in the `<security>` element of the `web-services.xml` deployment descriptor file to determine how it should update the WSDL.

B. Client Application

The client application loads the signature key pair and certificate from its client keystore and uses the `weblogic.webservice.context.WebServiceContext` API to add the public key and certificate as attributes to the Web Service session.

Note: The client application uses the key pair and certificate loaded from its client keystore to digitally sign the SOAP request. WebLogic Server later uses the key pair and certificate to encrypt the SOAP response.

C. WebLogic Web Services Client Runtime Environment

When the client application is executed, the Web Services client runtime environment, packaged in the client runtime JAR files, performs the following tasks:

Note: The client runtime performs all encryption and signature tasks directly before it sends the request to WebLogic Server and after all client handlers have executed.

1. Reads the WSDL of the Web Service being invoked to determine what parts of the SOAP request should be digitally signed or encrypted. The client runtime also gets the certificate for the server encryption key from the WSDL.
2. Generates the unsecured SOAP message request.
3. Creates a `<Security>` element in the header of the SOAP request that will contain the security information.
4. If required by the WSDL, inserts a username token with the client's username and password into the `<Security>` header of the SOAP request.
5. If the WSDL requires that the SOAP request be digitally signed, the Web Services client runtime environment:
   a. Generates a digital signature, according to the WSDL requirements, using the private key from the client's `WebServiceContext`.
   b. Adds the digital signature to the `<Security>` header of the SOAP request.
   c. Adds the certificate from the client's `WebServiceContext` to the `<Security>` header of the SOAP request. WebLogic Server later uses this certificate to verify the signature.
6. If the WSDL requires that the SOAP request be encrypted, the Web Services client runtime environment:
   a. Gets WebLogic Server's public encryption key from the certificate published in the WSDL.

b. Encrypts the SOAP request according to the requirements in the WSDL using WebLogic Server's public encryption key. The WSDL specifies what part of the SOAP message should be encrypted.

c. Adds a description of the encryption to the `<Security>` header of the SOAP request. WebLogic Server later uses this description to decrypt the SOAP request.

7. The Web Services client runtime sends the encrypted and signed SOAP request to WebLogic Server.

D. WebLogic Server

1. Receives the SOAP request and extracts the `<Security>` header.
2. If the SOAP request has been encrypted, WebLogic Server:
   a. Reads the description of the encryption and decrypts the SOAP request using WebLogic Server's private encryption key from its identity keystore.
   b. Removes the encryption description from the `<Security>` header.
3. If the SOAP request has been digitally signed, WebLogic Server performs the following tasks to verify the signature:
   a. Extracts the client's certificate from the `<Security>` header of the SOAP request.
   b. Extracts the digital signature from the `<Security>` header.
   c. Verifies the signature using the client's public key, obtained from the client's certificate.
   d. Asserts the identity of the client certificate to ensure that it maps to a valid WebLogic Server user.
   e. Removes the signature from the `<Security>` header.
4. Extracts, if present, the username token from the `<Security>` header.
5. Asserts the identity of the user and verifies their password. The rest of the invocation of the Web Service operation is run as this user.
6. Removes the `<Security>` header from the SOAP request.
7. Saves the client certificate that was included in the SOAP request for encrypting the SOAP response, if required.
8. Verifies that the specifications in the `<Security>` header matched the requirements in the WSDL.

9. Sends the post-processed SOAP request to the Web Services runtime for standard invocation.

When WebLogic Server sends the SOAP response back to the client, and it is required to digitally sign or encrypt the SOAP response, it follows the same steps as the WebLogic Web Services client runtime environment did when it initially sent its SOAP request (see "C. WebLogic Web Services Client Runtime Environment"), but with the following differences:

- WebLogic Server uses the public key from the saved client certificate to encrypt the SOAP response. The client application in turn uses the private key from its `WebServiceContext` (originally loaded from the client keystore) to decrypt the response.
- WebLogic Server uses the signature key pair and certificate from its identity keystore to digitally sign the SOAP response. WebLogic Server includes this certificate in the response so that the client application can verify the signature.
- WebLogic Server includes a username token in the SOAP response only if explicitly specified in the `web-services.xml` deployment descriptor file. Typically this is not needed because the client application does not need to assert the identity.

Configuring Message-Level Security: Main Steps

Configuring message-level security for a WebLogic Web Service involves some standard security tasks, such as obtaining digital certificates, creating keystores, and users, as well as Web Service-specific tasks, such as updating the `web-services.xml` file with security information.

To configure message-level security for a WebLogic Web Service and a client that invokes the service, follow these steps. Later sections describe some steps in more detail.

Note: The following procedure assumes that you have already implemented and assembled a WebLogic Web Service and you want to update it to use digital signatures and encryption.

1. Obtain two sets of key pair and digital certificates to be used by WebLogic Web Services. Although not required, BEA recommends that you obtain key pairs and certificates that will be used *only* by WebLogic Web Services.

   Warning: BEA requires that the key length be 1024 bits or larger.

   For clarity, it is assumed that the key pair/certificate used for digital signatures has a name `digSigKey` and password `digSigKeyPassword` and the one used for encryption has a name `encryptKey` and password `encryptKeyPassword`.

   You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

For details, see *Obtaining Private Keys and Digital Signatures* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html#get_keys_certs_trustedcas.

2. Create, if one does not currently exist, a custom identity keystore for WebLogic Server and load the key pairs and digital certificates you obtained in the preceding step into the identity keystore.

   If you have already configured WebLogic Server for SSL, then you have already created a identity keystore which you can also use for WebLogic Web Services data security purposes.

   You can use WebLogic's `ImportPrivateKey` utility and Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   For details, see *Creating a Keystore and Loading Key Pairs Into the Keystore* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html#keystore_creating.

3. Using the Administration Console, configure WebLogic Server to locate the keystore you created in the preceding step. If you are using a keystore that has already been configured for WebLogic Server, you do not need to perform this step.

   For details, see *Configuring Keystores* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html#ConfiguringKeystores.

4. Create a keystore used by the client application. BEA recommends that you create one client keystore per application user.

   You can use the Cert Gen utility or Sun Microsystem's `keytool` utility to perform this step. For development purposes, the `keytool` utility is the easiest way to get started.

   Later sections of this document assume you created a client keystore called `client_keystore` with password `client_keystore_password`.

   For details, see *Obtaining Private Keys and Digital Signatures* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html#get_keys_certs_trustedcas.

5. Create a key pair and a digital certificate, and load them into the client keystore. The same key pair will be used to digitally sign the SOAP request and encrypt the SOAP responses. The digital certificate will be mapped to a user of WebLogic Server, created in a later step.

   Warning: BEA requires that the key length be 1024 bits or larger.

   You can use Sun Microsystem's `keytool` utility to perform this step.

   Later sections of this document assume you created a key pair called `client_key` with password `client_key_password`.

6. Using the Administration Console, configure an Identity Asserter provider for your WebLogic Server security realm.

   WebLogic Server provides a default security realm, called `myrealm`, which is configured with a default Identity Asserter provider. Use this default security realm if you do not want to configure your own Identity Asserter provider. You must, however, perform additional configuration tasks to ensure that the default Identity Asserter Provider works correctly with message-secured WebLogic Web Services.

   For details, see "Configuring The Identity Asserter Provider for the myrealm Security Realm" on page 13-12.

7. Using the Administration Console, create users for authentication in your security realm.

   For details, see *Creating Users* at http://e-docs.bea.com/wls/docs81/secwlres/usrs_grps.html.

   Later sections of this guide assume you created a user `auth_user` with password `auth_user_password`.

8. Update the `build.xml` file that contains the call to the `servicegen` Ant task by adding the `<security>` child element to the `<service>` element that builds your Web Service. Specify information such as the encryption key pair, the digital signature key pair, and their corresponding passwords.

   Note: The `servicegen` Ant task offers only course-grained control of the encryption and digital signature configuration for a Web Service. For more fine-grained control of the data in the SOAP message that is encrypted or digitally signed, you must update the `web-services.xml` file manually. For details, see "Updating Security Information in the web-services.xml File" on page 13-14.

   For details about using `servicegen`, see "Updating the servicegen build.xml File" on page 13-12.

9. Re-run the `servicegen` Ant task to re-assemble your Web Service and regenerate the `web-services.xml` deployment descriptor.

10. Optionally encrypt the various passwords in the `web-services.xml` file of the EAR for your domain before deploying the EAR file to WebLogic Server. Typically you perform this step only when you deploy your Web Service in production mode.

    For details, see "Encrypting Passwords in the web-services.xml File" on page 13-21.

11. Update your client application to invoke the message-secured Web Service.

For details, see "Updating a Java Client to Invoke a Data-Secured Web Service" on page 13-23.

Configuring The Identity Asserter Provider for the myrealm Security Realm

You can use the default Identity Asserter provider, configured for the default myrealm security realm, with message-secured WebLogic Web Services. You must, however, perform some additional configuration tasks:

1. In the left pane of the Administration Console, expand the Security→Realms→myrealm→Providers→Authentication folder.
2. Click DefaultIdentityAsserter under the Authentication folder. The page to configure the default Identity Asserter appears in the right pane, open to the General tab.
3. In the right pane, scroll down to the Types box.
4. Move `X.509` from the Available box to the Chosen box.
5. Click Apply.
6. Select the Details tab.
7. Ensure that Use Default User Name Mapper is checked.
8. Select the Default User Name Mapper Attribute Type used when mapping the X.509 digital certificate to a user name.
9. Select the Default User Name Mapper Attribute Delimiter.
10. Click Apply.

For additional information about configuring the Identity Asserter, see:

- WebLogic Identity Asserter Provider -> Details at http://e-docs.bea.com/wls/docs81/ConsoleHelp/security_defaultidentityasserter_details.html
- WebLogic Identity Asserter Provider -> General at http://e-docs.bea.com/wls/docs81/ConsoleHelp/security_defaultidentityasserter_general.html

Updating the servicegen build.xml File

Update the `build.xml` file that contains the call to the `servicegen` Ant task by adding a `<security>` child element to the `<service>` element that builds your Web Service, as shown in the following example. By default, `servicegen` specifies that the *entire* SOAP body will be

digitally signed or encrypted, rather than specific elements. Later sections describe how to digitally sign or encrypt specific elements.

Note: For clarity, the following excerpt of `servicegen`'s `build.xml` file contains passwords in clear text. However, for security reasons, BEA recommends that you update your `build.xml` file to prompt for the passwords, using the `<input>` Ant task, rather than actually store the passwords in the file. For details on using the `<input>` Ant task, see *Apache Ant User Manual* at http://ant.apache.org/manual/.

```
<servicegen
   destEar="ears/myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="jars/myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True" >
       <security
               signKeyName="digSigKey"
               signKeyPass="digSigKeyPassword"
               encryptKeyName="encryptKey"
               encryptKeyPass="encryptKeyPassword"
       />
   </service>
</servicegen>
```

The preceding `build.xml` file specifies that `servicegen` assemble a Web Service that includes the following message-level security information in the `web-services.xml` deployment descriptor file:

- The `signKeyName` and `signKeyPass` attributes specify that the body of the SOAP request and response must be digitally signed. WebLogic Server uses the key pair and certificate, accessed using the name `digSigKey` and password `digSigKeyPassword`, from its keystore to digitally sign the SOAP response. The key pair and certificate are those that you added in step 1 of "Configuring Message-Level Security: Main Steps" on page 13-9.
- The `encryptKeyName` and `encryptKeyPass` attributes specify that the body of the SOAP request and response must be encrypted. WebLogic Server uses the key pair and certificate,

accessed using the name `encryptKey` and password `encryptKeyPassword`, from its keystore to encrypt and decrypt the SOAP request. The key pair and certificate are those that you added in step 1 of "Configuring Message-Level Security: Main Steps" on page 13-9.

Note: Always encrypt the passwords in the `web-services.xml` file with the `weblogic.webservice.encryptpass` utility, described in "Encrypting Passwords in the web-services.xml File" on page 13-21.

If you use the `<security>` element of the `servicegen` Ant task to add security to your Web Service, the entire SOAP body is encrypted and digitally signed for all operations of the Web Service. The encryption and digital signatures occur for both the request and response SOAP messages.

If you want more fine-grained control, such as specifying particular elements of the SOAP message to be digitally signed or encrypted, a subset of operations that have message-level security, and so on, update the `web-services.xml` file of your WebLogic Web Service manually. For details, see "Updating Security Information in the web-services.xml File" on page 13-14.

Updating Security Information in the web-services.xml File

The `servicegen` Ant task adds minimal default message-level security information to the generated `web-services.xml` deployment descriptor file. In particular, the default information specifies that, for all operations of the Web Service, the *entire* body of the SOAP messages be digitally signed or encrypted, rather than specific elements. This default behavior is adequate in many cases; however, you might sometimes want to specify just a subset of the elements to be digitally signed or encrypted, as well as specify different security specifications for different operations. In this case, you must update the `web-services.xml` file manually.

If you use the `build.xml` file in "Updating the servicegen build.xml File" on page 13-12 to run `servicegen`, the following example shows the resulting `<security>` element in the generated `web-services.xml` file; the sections in bold are described after the example:

```
<web-service>
...
  <security>

    <signatureKey>
      <name>digSigKey</name>
      <password>digSigKeyPassword</password>
    </signatureKey>
```

```
    <encryptionKey>
      <name>encryptKey</name>
      <password>encryptKeyPassword</password>
    </encryptionKey>

    <spec:SecuritySpec xmlns:spec="http://www.openuri.org/2002/11/wsse/spec"
                       Namespace="http://schemas.xmlsoap.org/ws/2002/07/secext"
                       Id="default-spec">

      <spec:BinarySecurityTokenSpec
                      xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
                      EncodingType="wsse:Base64Binary"
                      ValueType="wsse:X509v3">
      </spec:BinarySecurityTokenSpec>

      <spec:SignatureSpec
             SignatureMethod="http://www.w3.org/2000/09/xmldsig#rsa-sha1"
             SignBody="true"
             CanonicalizationMethod="http://www.w3.org/2001/10/xml-exc-c14n#">
      </spec:SignatureSpec>

      <spec:EncryptionSpec
            EncryptBody="true"
            EncryptionMethod="http://www.w3.org/2001/04/xmlenc#tripledes-cbc">
      </spec:EncryptionSpec>

    </spec:SecuritySpec>

  </security>

  ...

  <operations>
      <operation
             name="myOperation" method="myMethod" component="ejbComp"
             in-security-spec="default-spec" out-security-spec="default-spec">
      ...
      </operation>
  </operations>

  ...

</web-service>
```

Note: The `spec` prefix in the preceding example is a namespace prefix that is required for the security information in the `web-services.xml` deployment descriptor file. For more information about XML namespaces, see Namespaces in XML at http://www.w3.org/TR/REC-xml-names/.

The <signatureKey> and <encryptKey> elements in the preceding web-services.xml excerpt specify the username and passwords used to retrieve the keys for digital signatures and encryption, respectively, from the server's keystore.

The Id="default-spec" attribute of the <spec:SecuritySpec> element specifies that it is the default security specification. By default, the SOAP requests and responses for invokes of *all* operations of the Web Service must follow the security information described by this security specification; this is specified with the in-security-spec="default-spec" and out-security-spec="default-spec" attributes of each <operation> element.

The SignBody="true" and EncryptBody="true" attributes of the <spec:SignatureSpec> and <spec:EncryptionSpec> elements specify that the entire body of the SOAP messages for all operations must be digitally signed and encrypted.

The following sections describe how to update the web-services.xml file to specify more fine-grained message-level security:

- "Digitally Signing or Encrypting a Particular Element in the SOAP Message" on page 13-16
- "Associating an Operation with a Particular Security Specification" on page 13-17
- "Using Timestamps" on page 13-19

Digitally Signing or Encrypting a Particular Element in the SOAP Message

To specify particular elements to be digitally signed or encrypted, add one or more <spec:ElementIdentifier> child elements to the <spec:SignatureSpec> or <spec:EncryptionSpec> element, respectively, in the web-services.xml file.

For example, assume that, in addition to the entire SOAP body, you want to digitally sign an element in the SOAP header whose local name is Timestamp. To specify this configuration, add a <spec:ElementIdentifier> child element to the <spec:SignatureSpec> element as shown:

```
<spec:SignatureSpec
         SignatureMethod="http://www.w3.org/2000/09/xmldsig#rsa-sha1"
         SignBody="true"
         CanonicalizationMethod="http://www.w3.org/2001/10/xml-exc-c14n#">

      <spec:ElementIdentifier
              LocalPart="Timestamp"
              Namespace="http://www.bea.com/examples/security" />
</spec:SignatureSpec>
```

The example shows how to identify that the `Timestamp` element of the SOAP message be digitally signed by using the `LocalPart` and `Namespace` attributes of the `<spec:ElementIdentifier>` element. Set the `LocalPart` attribute equal to the name of the element in the SOAP message you want to encrypt and the `Namespace` attribute to its namespace. To get the exact name and namespace of the element, you can:

- Look at the WSDL of the Web Service. To get the WSDL of a WebLogic Web Service, use the `wsdlgen` Ant task on the existing non-secure Web Service. For details, see "wsdlgen" on page B-50.
- View the actual SOAP messages generated from an invoke of the operation when deployed as a non-secure Web Service. For details, see "Using the Web Service Home Page to Test Your Web Service" on page 20-2.

Specifying a particular element to be encrypted is very similar. For example, to encrypt just the element `CreditCardNumber`, wherever it appears in the SOAP message (rather than the entire SOAP body), update the `<spec:EncryptionSpec>` element as shown:

```
<spec:EncryptionSpec
     EncryptionMethod="http://www.w3.org/2001/04/xmlenc#tripledes-cbc" >

       <spec:ElementIdentifier
              LocalPart="CreditCardNumber"
              Namespace="http://www.bea.com/examples/security" />
</spec:EncryptionSpec>
```

For details about the `<security>` element, and all its child elements discussed in this section, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

Associating an Operation with a Particular Security Specification

The `<security>` element of the `web-services.xml` deployment descriptor file can contain zero or more `<spec:SecuritySpec>` elements. These elements specify the security requirements for a particular SOAP message: what should be signed, what should be encrypted, what tokens should be included, and so on.

Each `<spec:SecuritySpec>` element typically has an `Id` attribute that uniquely identifies it. In the `<operations>` section of the `web-services.xml` file, each `<operation>` element can reference a specific security specification by setting the operation's `in-security-spec` or `out-security-spec` attribute to the relevant `Id` value. The security specification referenced by the `in-security-spec` attribute is applied to SOAP requests; the security specification referenced by the `out-security-spec` attribute is applied to SOAP responses.

If a `<spec:SecuritySpec>` element contains no `Id` attribute, or it is assigned the value `default-spec`, the security specification is treated as the default specification and is applied to all operations that do not explicitly reference a specification. Only one default specification can be defined: if more than one is defined in the `web-services.xml` file, the Web Service will not deploy.

The `servicegen` Ant task always generates a default security specification in the generated `web-services.xml` file (with an `Id="default-spec"` attribute) and this security specification is applied to all SOAP messages for all operations. The individual `<operation>` elements do not contain any direct reference to this security specification, since none is needed.

For example, assume you have defined the following two security specifications for a Web Service:

```
<web-service>
...
  <security>
  ...
  <spec:SecuritySpec xmlns:spec="http://www.openuri.org/2002/11/wsse/spec"
                     Namespace="http://schemas.xmlsoap.org/ws/2002/07/secext"
                     Id="encrypt-only">
       <spec:EncryptionSpec
            ...
       </spec:EncryptionSpec>
  </spec:SecuritySpec>

  <spec:SecuritySpec xmlns:spec="http://www.openuri.org/2002/11/wsse/spec"
                     Namespace="http://schemas.xmlsoap.org/ws/2002/07/secext"
                     Id="sign-only">
      <spec:SignatureSpec
          ...
      </spec:SignatureSpec>
  </security>
  ...
</web-service>
```

In the example, the `encrypt-only` security specification requires only encryption and the `sign-only` security specification requires only digital signatures. You can mix and match these security specifications for particular operations by using the `in-security-spec` and `out-security-spec` attributes of the relevant `<operation>` element, as shown in the following example:

```
  <operations>
      <operation
             name="operationOne" method="methodOne" component="ejbComp"
             in-security-spec="encrypt-only"
```

```
            out-security-spec="encrypt-only">
      ...
      </operation>
      <operation
            name="operationTwo" method="methodTwo" component="ejbComp"
            in-security-spec="sign-only">
      ...
      </operation>
</operations>
```

The preceding excerpt shows that both the SOAP request and response of the `operationOne` operation must be encrypted, but not digitally signed. The SOAP request for `operationTwo` must be digitally signed (although not encrypted), but the SOAP response requires no security at all.

For details about the `<security>` and `<operation>` elements, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

Using Timestamps

When a client application invokes a WebLogic Web Service that has been configured for message-level security, WebLogic Server may also require and add timestamp information in the SOAP request and response. By default, WebLogic Server:

- Requires that digitally signed SOAP requests include a timestamp and rejects any that do not. The timestamp itself must be digitally signed.
- Assumes that its clock and the client application's clock are *not* synchronized. This means that if the SOAP request from a client application includes a timestamp with an expiration, WebLogic Server rejects the message. This is because WebLogic Server is unable to ensure that the message has not already expired.
- Adds a timestamp to the SOAP response. The timestamp contains only the creation date of the SOAP response; it does not contain an expiration date.

You can change the default timestamp behavior of your WebLogic Web Service by adding a `<timestamp>` child element to the `<security>` element in the `web-services.xml` deployment descriptor.

The following `web-services.xml` excerpt shows an example of configuring timestamp behavior:

```
<web-service>
 ...
  <security>
    <timestamp>
```

```
        <clocks-synchronized>true</clocks-synchronized>
        <clock-precision>30000</clock-precision>
        <require-signature-timestamp>false</require-signature-timestamp>
        <generate-signature-timestamp>true</generate-signature-timestamp>
        <inbound-expiry>120000</inbound-expiry>
        <outbound-expiry>30000</outbound-expiry>
      </timestamp>
      ...
    </security>
    ...
</web-service>
```

The preceding `<timestamp>` element specifies the following timestamp behavior when the relevant Web Service operation is invoked:

- WebLogic Server's and the client application's clocks are synchronized, and are accurate within 30000 milliseconds (30 seconds) of each other. This implies that if WebLogic Server receives a digitally signed SOAP request whose expiration period is less than 30 seconds, WebLogic Server automatically rejects the request because it cannot accurately determine if the message has expired.
- WebLogic Server does not require that digitally signed SOAP requests include a timestamp, although it always includes a timestamp in its digitally signed SOAP responses.
- WebLogic Server has its own expiration period for digitally signed SOAP requests of 120000 milliseconds (2 minutes). This means that, independent of any client-specified expiration, WebLogic Server rejects the request if it receives it more than 2 minutes after it was created (adjusting for clock precision.)
- WebLogic Server includes an expiration of 30000 milliseconds (30 seconds) in the SOAP response.

The value specified for the `<clock-precision>` element is a reflection of how accurately the clocks are synchronized between WebLogic Server and the client applications that invoke the Web Service operation. WebLogic Server uses the value to round all timestamps in a consistent manner. For example, assume that the clock precision is 30000 milliseconds, or 30 seconds. This means that all timestamps are rounded to the closest 30 second increment. This means that, in this example, WebLogic Server rounds the times `12:00:10` and `11:59:50` to the same time (`12:00:00`) and thus treats the two timestamps equally.

Each of the timestamp elements of the `web-services.xml` deployment descriptor has a client-side equivalent system property that you can set in your client application. For details, see "Using Web Services System Properties" on page 7-14.

For detailed descriptions of the `<timestamp>` element and all its child elements, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

Encrypting Passwords in the web-services.xml File

Encrypt the key pair passwords (used for encryption and digital signatures) in the `web-services.xml` file with the `weblogic.webservice.encryptpass` utility.

The `weblogic.webservice.encryptpass` utility updates the specified EAR file (or exploded directory) by editing the `<security>` element of the `web-services.xml` file, replacing any plain text passwords with their encrypted equivalents.

Only the WebLogic Server domain you specify to the utility is able to decrypt the passwords. This means that if, for example, you want to deploy the EAR file on a WebLogic Server domain different from the one you specified in the `encryptpass` utility, you must rerun the utility against the EAR file that contains plain text passwords, specifying the new domain.

To encrypt the passwords:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

2. Change to the domain directory of the WebLogic Server domain which will be deploying your EAR file. The domain directory contains the `config.xml` file for the WebLogic Server.

   Warning: Only this WebLogic Server domain will be able to decrypt the encrypted passwords in the `web-services.xml` file.

3. Run the utility:

   ```
   java weblogic.webservice.encryptpass options ear_or_dir
   ```

where

– *options* refers to one or more of the options described in Table 13-1.

– *ear_or_dir* refers to the full path name of the EAR file (or exploded directory) for which you want to encrypt the passwords in the `web-services.xml` file.

The following example shows how to encrypt the passwords for the `Hello` Web Service packaged in the `ears/myService.ear` file:

```
java weblogic.webservice.encryptpass -serviceName Hello -verbose
ears/myService.ear
```

Table 13-1 Options for the weblogic.webservice.encryptpass Utility

Option	Description
`-help`	Prints the usage message for the utility.
`-version`	Prints the version information for the utility.
`-verbose`	Enables verbose output.
`-warName` *name*	Specifies the name of the Web application WAR file, packaged inside the EAR file, that contains the `web-services.xml` file. Default value is `web-services.war`.
`-serviceName` *name*	Specifies the name of the Web Service for which you want to encrypt passwords. The name corresponds to the `name` attribute of the Web Service's `<web-service>` element in the `web-services.xml` file. Default value is the first Web Service in the `web-services.xml` file.
`-domain` *directory*	Specifies the domain directory of the WebLogic Server to which you want to deploy your Web Service. Default value is the current directory from which you are running the utility.

Updating a Java Client to Invoke a Data-Secured Web Service

To update a Java client application to invoke either a WebLogic or a non-WebLogic Web Service that uses digital signatures or encryption:

1. Update your client application's CLASSPATH to include *WL_HOME*/server/lib/wsse.jar, where *WL_HOME* refers to the top-level directory of WebLogic Platform. This client JAR file contains BEA's implementation of the Web Services Security (WS-Security) specification.
2. Update your Java code to load a key pair and digital certificate from the client's keystore and pass this information, along with a username and password for user authentication, to the secure WebLogic Web Service being invoked.

   For details, see "Writing the Java Code to Invoke a Secure WebLogic Web Service" on page 13-23.

   For an example of invoking a secure non-WebLogic Web Service, see "Writing the Java Code to Invoke a Secure Non-WebLogic Web Service" on page 13-26
3. Run the client application.

   For details about system properties you can set to get more information about the digital signatures and encryption, see "Running the Client Application" on page 13-31.

Writing the Java Code to Invoke a Secure WebLogic Web Service

The following example shows a Java client application that invokes a message-secured WebLogic Web Service, with the security-specific code in bold (and described after the example):

```
import java.io.IOException;
import java.io.FileInputStream;

import javax.xml.rpc.ServiceException;

import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.CertificateException;
import java.security.UnrecoverableKeyException;
import java.security.Key;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.cert.X509Certificate;

import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.context.WebServiceSession;
import weblogic.webservice.core.handler.WSSEClientHandler;
```

```
import weblogic.xml.security.UserInfo;

public class Main{

  private static final String CLIENT_KEYSTORE = "client_keystore";
  private static final String KEYSTORE_PASS = "client_keystore_password";
  private static final String CLIENT_KEYNAME = "client_key";
  private static final String CLIENT_KEYPASS = "client_key_password";
  private static final String AUTHENTICATION_USER = "auth_user";
  private static final String AUTHENTICATION_USER_PASS = "auth_user_password";

  public static void main( String[] args ){

    if( args.length == 1 ){
      new Main( args[0] );
    }else{
      throw new IllegalArgumentException( "URL of the service not specified" );
    }
  }

  public Main( String wsdlUrl ){

    try{
      HelloWorldService service = new HelloWorldService_Impl( wsdlUrl );
      HelloWorldServicePort port = service.getHelloWorldServicePort();

      WebServiceContext context = service.context();

      X509Certificate clientcert = getCertificate(CLIENT_KEYNAME,
CLIENT_KEYSTORE);

      PrivateKey clientprivate = (PrivateKey)getPrivateKey(CLIENT_KEYNAME,
CLIENT_KEYPASS,CLIENT_KEYSTORE);

      WebServiceSession session = context.getSession();

      session.setAttribute(WSSEClientHandler.CERT_ATTRIBUTE, clientcert);
      session.setAttribute(WSSEClientHandler.KEY_ATTRIBUTE, clientprivate);

     UserInfo ui = new UserInfo(AUTHENTICATION_USER, AUTHENTICATION_USER_PASS);
      session.setAttribute(WSSEClientHandler.REQUEST_USERINFO, ui);

      World world = port.helloComplexWorld();

      System.out.println( world );

    }catch( IOException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( ServiceException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( KeyStoreException e ){
      System.out.println( "Failed to create web service client:" + e );
```

```
    }catch( CertificateException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( UnrecoverableKeyException e ){
      System.out.println( "Failed to create web service client:" + e );
    }catch( NoSuchAlgorithmException e ){
      System.out.println( "Failed to create web service client:" + e );
    }
  }

  private Key getPrivateKey( String keyname, String password, String keystore)
      throws IOException, KeyStoreException, NoSuchAlgorithmException,
             CertificateException, UnrecoverableKeyException{

    KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream(keystore), KEYSTORE_PASS.toCharArray());
    Key result = ks.getKey(keyname, password.toCharArray());
    return result;
  }

  private static X509Certificate getCertificate(String keyname, String keystore)
     throws IOException, KeyStoreException, NoSuchAlgorithmException,
            CertificateException {

    KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream(keystore), KEYSTORE_PASS.toCharArray());
    X509Certificate result = (X509Certificate) ks.getCertificate(keyname);
    return result;
  }

}
```

The main points to note about the preceding code are:

- Once you create the JAX-RPC `Service` object, get the WebLogic Web Service context:

  ```
  WebServiceContext context = service.context();
  ```

  Note: The `weblogic.webservice.context.WebServiceContext` class is a proprietary WebLogic Web Service client API.

- Load the needed key pairs and X.509 digital certificates from a client keystore:

  ```
  X509Certificate clientcert =
         getCertificate(CLIENT_KEYNAME, CLIENT_KEYSTORE);

  PrivateKey clientprivate =
        (PrivateKey)getPrivateKey(CLIENT_KEYNAME,
  CLIENT_KEYPASS,CLIENT_KEYSTORE);
  ```

- From the WebLogic Web Service context, get the session information:

  ```
  WebServiceSession session = context.getSession();
  ```

Note: The `weblogic.webservice.context.WebServiceSession` class is a WebLogic Web Service client API.

- Use `WebServiceSession` attributes to pass the private key and digital certificates to the WebLogic Web Service being invoked:

```
session.setAttribute(WSSEClientHandler.CERT_ATTRIBUTE, clientcert);
session.setAttribute(WSSEClientHandler.KEY_ATTRIBUTE,
clientprivate);
```

- Create a `UserInfo` object that contains the authentication username and password, and use an attribute of the `WebServiceSession` to pass the information to the WebLogic Web Service being invoked:

```
UserInfo ui = new UserInfo(AUTHENTICATION_USER,
AUTHENTICATION_USER_PASS);
session.setAttribute(WSSEClientHandler.REQUEST_USERINFO, ui);
```

Note: The `weblogic.xml.security.UserInfo` class is a WebLogic Web Service client API.

- The local methods `getPrivateKey()` and `getCertificate()` are simple examples of how to get information from the client's local keystore. Depending on how you have set up your client keystore, you will use different ways of getting this information.

For more information about the WebLogic Web Services APIs discussed in this section, see the Javadoc.

Writing the Java Code to Invoke a Secure Non-WebLogic Web Service

The following example is similar to the one in the previous section, except that it shows how to write a Java client application that invokes a non-WebLogic Web Service, such as .NET.

The example uses the `weblogic.xml.security.wsse` and `weblogic.xml.security.specs` APIs to create user `Tokens`, X.509 `Tokens`, `EncryptionSpecs`, and `SignatureSpecs` which the WebLogic client API uses to create the appropriate `<wsse:Security>` element in the SOAP message request that invokes the non-WebLogic Web Service. The user `Token` objects contain username and passwords and X.509 `Token` objects contain a certificate and an optional private key.

Note: Because there is currently no standard way of specifying security information in the WSDL of a Web Service, consult with the Web Service provider to find out what needs to be signed and encrypted when invoking a non-WebLogic Web Service.

The relevant sections of the example are in bold (and described after the example):

```
import java.io.IOException;
import java.io.FileInputStream;

import java.util.List;
import java.util.ArrayList;
import java.security.KeyStore;
import java.security.PrivateKey;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateException;

import javax.xml.rpc.ServiceException;

import javax.xml.namespace.QName;

import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.HandlerRegistry;

import weblogic.webservice.context.WebServiceContext;
import weblogic.webservice.core.handler.WSSEClientHandler;
import weblogic.xml.security.wsse.Security;
import weblogic.xml.security.wsse.Token;
import weblogic.xml.security.wsse.SecurityElementFactory;
import weblogic.xml.security.specs.EncryptionSpec;
import weblogic.xml.security.specs.SignatureSpec;
import weblogic.xml.security.SecurityAssertion;
import examples.security.basicclient.BasicPort;
import examples.security.basicclient.Basic_Impl;
import examples.security.basicclient.Basic;

public class Client {

  private static final String CLIENT_KEYSTORE = "client.keystore";
  private static final String KEYSTORE_PASS = "gumby1234";

  private static final String KEY_ALIAS = "joe";
  private static final String KEY_PASSWORD = "myKeyPass";

  private static final String SERVER_KEY_ALIAS = "myServer";

  private static final String USERNAME = "pete";
  private static final String USER_PASSWORD = "myPassword";

  public static void main(String[] args)
      throws IOException, ServiceException, Exception {

    {
      final KeyStore keystore = loadKeystore(CLIENT_KEYSTORE, KEYSTORE_PASS);
```

```
Basic service = new Basic_Impl();
WebServiceContext context = service.context();

// add WSSE Client Handler to the handler chain for the service.
HandlerRegistry registry = service.getHandlerRegistry();

List list = new ArrayList();

list.add(new HandlerInfo(WSSEClientHandler.class, null, null));

registry.setHandlerChain(new QName("basicPort"), list);

// load the client credential
X509Certificate clientcert;
clientcert = getCertificate(KEY_ALIAS, keystore);

PrivateKey clientprivate;
clientprivate = getPrivateKey(KEY_ALIAS, KEY_PASSWORD, keystore);

// load the server's certificate...
X509Certificate serverCert = getCertificate(SERVER_KEY_ALIAS, keystore);

// configure the Security element for the service.
SecurityElementFactory factory =
          SecurityElementFactory.getDefaultFactory();

Token x509token = factory.createToken(clientcert, clientprivate);
Token userToken = factory.createToken(USERNAME, USER_PASSWORD);

EncryptionSpec encSpec = EncryptionSpec.getDefaultSpec();
SignatureSpec sigSpec = SignatureSpec.getDefaultSpec();

Token serverToken = null;

// create a token for the server's cert... no PrivateKey...
serverToken = factory.createToken(serverCert, null);

Security security = factory.createSecurity(/* role */ null);

//add a Timestamp to the Security header. The creation time will be
// the current time, and there is no expiration.
security.addTimestamp();

//add the username/password to the header as a UsernameToken
security.addToken(userToken);

security.addSignature(x509token, sigSpec);

//add client cert for signature verification and response encryption
// should be added after the signature....
security.addToken(x509token);
```

```
      security.addEncryption(serverToken, encSpec);

      BasicPort port = service.getbasicPort();

      // add the security element to the request...
      context.getSession().setAttribute("weblogic.webservice.security.request",
                                        security);

      String result = null;
      result = port.helloback();

      System.out.println(result);

      // view the assertions from processing the server's response...
      SecurityAssertion[] assertions = (SecurityAssertion[])

context.getSession().getAttribute("weblogic.webservice.security.assertions.res
ponse");
      for (int i = 0; i < assertions.length; i++) {

        SecurityAssertion assertion = assertions[i];
        System.out.println(assertion);
      }
    }
  }

  private static KeyStore loadKeystore(String filename, String password)
      throws KeyStoreException, IOException, NoSuchAlgorithmException,
              CertificateException {
    final KeyStore ks = KeyStore.getInstance("JKS");
    ks.load(new FileInputStream(filename), password.toCharArray());
    return ks;
  }

  private static PrivateKey getPrivateKey(String alias, String password,
         KeyStore keystore)
      throws Exception {
    PrivateKey result =
       (PrivateKey) keystore.getKey(alias, password.toCharArray());

    return result;
  }

  private static X509Certificate getCertificate(String alias, KeyStore keystore)
      throws Exception {
    X509Certificate result = (X509Certificate) keystore.getCertificate(alias);
    return result;
  }
}
```

The main points to note about the preceding code are:

- Add the WSSE client handler to the client's handler chain:

```
HandlerRegistry registry = service.getHandlerRegistry();
List list = new ArrayList();
list.add(new HandlerInfo(WSSEClientHandler.class, null, null));
registry.setHandlerChain(new QName("basicPort"), list);
```

- Use the `weblogic.xml.security.wsse.SecurityElementFactory` to create an object that represents the `<wsse:Security>` element of the SOAP message. Also use this factory to create the user and X.509 tokens, as shown in the following code excerpts:

```
SecurityElementFactory factory =
          SecurityElementFactory.getDefaultFactory();

Token x509token = factory.createToken(clientcert, clientprivate);
Token userToken = factory.createToken(USERNAME, USER_PASSWORD);

Token serverToken = null;

Security security = factory.createSecurity(/* role */ null);
```

- Once you have the security object and tokens, create optional `EncryptionSpec` and `SignatureSpec` objects that specify the elements of the SOAP message that you want to encrypt or digitally sign, respectively.

```
EncryptionSpec encSpec = EncryptionSpec.getDefaultSpec();
SignatureSpec sigSpec = SignatureSpec.getDefaultSpec();
```

- Use the `addTimestamp()` method to to add a timestamp, and optional expiration date, to the security element in the SOAP message. Use one of the following four flavors of the `Security.addTimestamp()` method:
  - `addTimestamp()`—Sets the creation timestamp to the current time, with no expiration date.
  - `addTimestamp(long)`—Sets the creation timestamp to the current time and the expiration date to `long` number of milliseconds after the creation timestamp.
  - `addTimestamp(java.util.Calendar)`—Sets the creation timestamp to the value of the `Calendar` parameter, with no expiration date.
  - `addTimestamp(java.util.Calendar, java.util.Calendar)`—Sets the creation timestamp to the value of the first `Calendar` parameter and the expiration date to the value of the second `Calendar` parameter.
- Use the `addToken()`, `addSignature()`, and `addEncryption()` methods to add the tokens to the security element and to specify whether you want the SOAP message to be encrypted or digitally signed. If you created the optional `EncryptionSpec` or

`SignatureSpec` objects, specify them as parameters to the respective methods. If you do not specify these specs, the entire SOAP message body is encrypted or digitally signed.

```
security.addToken(userToken);
security.addSignature(x509token, sigSpec);
security.addToken(x509token);
security.addEncryption(serverToken, encSpec);
```

- Add the security element to the SOAP request by setting it as an attribute to the session using the `weblogic.webservice.security.request` attribute:

```
context.getSession().setAttribute("weblogic.webservice.security.request",
                                  security);
```

Keep the following points in mind when using the WebLogic Web Services Security APIs to invoke a secure non-WebLogic Web Service:

- When you use the `addXXX()` methods to add tokens and encryption and signature information to the `<wsse:Security>` element of the SOAP message, they are applied to the message in the order you specify. They appear, however, in *reverse* order in the resulting SOAP message.
- If you specify that you want the SOAP message to be digitally signed, in your Java code add the X.509 token used for the signature *after* you have added the signature using the appropriate `addXXX()` method. This is because the X.509 token, which specifies the actual certificate and optional private key, should be read by the recipient of the message *before* it processes the signature.
- You cannot encrypt or sign the contents of the `<wsse:Security>` element itself.

Running the Client Application

When you run the client application that uses digital signatures and encryption to invoke a Web Service, you can set the following system properties to view more runtime security information:

- `weblogic.xml.encryption.verbose=true`
- `weblogic.xml.signature.verbose=true`

Configuring Transport-Level Security (SSL): Main Steps

Transport-level security refers to securing the connection between a client application and a Web Service with Secure Sockets Layer (SSL). The following procedure describes the high-level steps; later sections in the chapter describe the steps in more detail.

1. Configure SSL for WebLogic Server.

   You can configure one-way SSL (the default) where WebLogic Server is required to present a certificate to the client application, or two-way SSL where both the client applications and WebLogic server present certificates to each other.

   For details about SSL, the difference between one-way and two-way, and procedures to configure both, see *Configuring SSL* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html.

2. Optionally update the `web-services.xml` file to specify that the Web Service can be accessed *only* by HTTPS.

   See "Specifying the HTTPS Protocol" on page 13-44.

3. Configure SSL for the client application.

   See "Configuring SSL for a Client Application" on page 13-32.

Warning: If you use two-way SSL to secure the connection when invoking a WebLogic Web Service, WebLogic Server always asserts the identity of the certificate to ensure that it maps to a valid WebLogic Server user, even if the Web Service or the stateless EJB back-end component does not require any special privileges.

WebLogic Server does not assert the identity of the certification in one-way SSL, however, because in that case the client application does not send its certificate.

Configuring SSL for a Client Application

Configure SSL for your client application by using either:

- The WebLogic Server-provided SSL implementation. See "Using the WebLogic Server-Provided SSL Implementation" on page 13-33.
- A third-party SSL implementation. See "Using a Third-Party SSL Implementation" on page 13-38.

If you are using two-way SSL, your client application must also present its certificate to WebLogic Server. For details, see "Configuring Two-Way SSL For a Client Application" on page 13-40.

For additional detailed information about the APIs discussed in this section see the Web Service security Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/webservice/client/package-summary.html.

Using the WebLogic Server-Provided SSL Implementation

If you are using a stand-alone client application, WebLogic Server provides an implementation of SSL in the `webserviceclient+ssl.jar` client runtime JAR file. In addition to the SSL implementation, this client JAR file contains the standard client JAX-RPC runtime classes contained in `webservicesclient.jar`.

Note: For information about BEA's current licensing of client functionality, see the BEA eLicense Web Site at http://elicense.bea.com/elicense_webapp/index.jsp.

To configure basic SSL support for your client application, follow these steps:

1. Set the filename of the file containing trusted Certificate Authority (CA) certificates. Do this by either:
   - Setting the `System` property `weblogic.webservice.client.ssl.trustedcertfile` to the name of the file that contains a collection of PEM-encoded certificates.
   - Executing the `BaseWLSSLAdapter.setTrustedCertificatesFile(String` *`ca_filename`*`)` method in your client application.
2. Run your Java client application, either as a standalone client or on WebLogic Server.

   If you are creating a standalone client application:
   - Add the *`WL_HOME`*`/server/lib/webserviceclient+ssl.jar` runtime Java client JAR file to you CLASSPATH, where *`WL_HOME`* refers to the top-level directory of WebLogic Platform. This client JAR file contains the client runtime implementation of JAX-RPC as well as the implementation of SSL.

     If your client application is running on WebLogic Server, you do not need this runtime client JAR file.
   - Set the following `System` properties on the command line:
     - `bea.home=`*`license_file_directory`*
     - `java.protocol.handler.pkgs=com.certicom.net.ssl`

     where *`license_file_directory`* refers to the directory that contains the BEA license file `license.bea`, as shown in the following example:

```
java -Dbea.home=/bea_home \
   -Djava.protocol.handler.pkgs=com.certicom.net.ssl  my_app
```

Note: If your client application is running on a computer different from the computer hosting WebLogic Server (which is typically the case), copy the BEA license file

from the server computer to a directory on the client computer, and then point the `bea.home` System property to this client-side directory.

3. If you are not using a certificate issued by a CA in your trusted CA file, then disable strict certificate validation by either setting the `weblogic.webservice.client.ssl.strictcertchecking` System property to `false` at the command line when you run the standalone application, or programmatically use the `BaseWLSSLAdapter.setStrictCheckingDefault()` method. Use the second way if your client application is running on WebLogic Server.

By default, client applications that use the WebLogic SSL implementation do not share sockets. If you want to change this behavior, see "Using SSL Socket Sharing When Using the WebLogic SSL Implementation" on page 13-36.

For detailed information, see the Web Service security Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/webservice/client/package-summary.html.

Configuring the WebLogic SSL Implementation Programatically

You can also configure the WebLogic Server-provided SSL implementation programatically by using the `weblogic.webservice.client.WLSSLAdapter` adapter class. This adapter class hold configuration information specific to WebLogic Server's SSL implementation and allows the configuration to be queried and modified.

The following excerpt shows an example of configuring the `WLSSLAdapter` class for a specific WebLogic Web Service; the lines in bold are discussed after the example:

```
// instantiate an adapter...
WLSSLAdapter adapter  = new WLSSLAdapter();
adapter.setTrustedCertifcatesFile("mytrustedcerts.pem");

// optionally set the Adapter factory to use this
// instance always...
SSLAdapterFactory.getDefaultFactory().setDefaultAdapter(adapter);
SSLAdapterFactory.getDefaultFactory().setUseDefaultAdapter(true);

//create service factory
ServiceFactory factory = ServiceFactory.newInstance();

//create service
Service service = factory.createService( serviceName );

//create call
Call call = service.createCall();

call.setProperty("weblogic.webservice.client.ssladapter",
                 adapter);
```

```
try {

  //invoke the remote web service
  String result = (String) call.invoke( new Object[]{ "BEAS" } );
  System.out.println( "Result: " +result);
} catch (JAXRPCException jre) {
...
}
```

The example first shows how to instantiate an instance of the WebLogic Server-provided `WLSSLAdapter` class, which supports the SSL implementation contained in the `webserviceclient+ssl.jar` file. It then configures the adapter instance by setting the name of the file that contains the Certificate Authority certificates using the `setTrustedCertificatesFile(String)` method; in this case the file is called `mytrustedcerts.pem`.

The example then shows how to set `WLSSLAdapter` as the default adapter of the adapter factory and configures the factory to always return this default.

Note: This step is optional; it allows *all* Web Services to share the same adapter class along with its associated configuration.

You can also set the adapter for a particular Web Service port or call. The preceding example shows how to do this when using the `Call` class to invoke a Web Service dynamically:

```
call.setProperty("weblogic.webservice.client.ssladapter", adapter);
```

Set the property to an object that implements the `weblogic.webservice.client.SSLAdapter` interface (which in this case is the WebLogic Server-provided `WLSSLAdapter` class.)

The following excerpt shows how to set the adapter when using the Stub interface to statically invoke a Web Service:

```
((javax.xml.rpc.Stub)stubClass)._setProperty("weblogic.webservice.client.sslad
apter", adapterInstance);
```

You can get the adapter for a specific instance of a Web Service call or port by using the following method for dynamic invocations:

```
call.getProperty("weblogic.webservice.client.ssladapter");
```

Use the following method for static invocations:

```
((javax.xml.rpc.Stub)stubClass)._getProperty("weblogic.webservice.client.sslad
apter");
```

For detailed information, see the Web Service security Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/webservice/client/package-summary.html.

Using SSL Socket Sharing When Using the WebLogic SSL Implementation

By default, socket sharing is disabled for SSL client applications that connect to a WebLogic Web Service using the WebLogic Server-provided SSL implemenation.

However, to improve the performance of your client application, you can enable socket sharing for multiple serial invokes of a Web Service. This socket sharing mechanism provides the improved performance of SSL connection reuse, while giving you the ability to enforce any necessary security.

Implications of Enabling SSL Socket Sharing

If your application is actually a server in which multiple clients use SSL authentication to invoke a Web Service, it is your responsibility to prevent access by one client to another client's JAX-RPC stub implementation object (`weblogic.webservice.core.rpc.StubImpl`).

Additionally, you must be aware of the following thread safety issues. The BEA generated JAX-RPC client stubs are thread-safe. However, as soon as you enable SSL socket sharing, the client stubs are no longer thread-safe. To minimize the chances of your Web Service client applications running into threading problems, BEA recommends you do either of the following:

- Implement your Web Service client as an EJB, either stateless session of message-driven. Then, if you want to use a single `weblogic.webservice.core.rpc.StubImpl` object for all operation invocations, the EJB container will prevent more that one WebLogic execute thread from running at a time. To do this, create an instance variable in the EJB to store the object that extends `StubImpl`, as shown in the following code snippet:

  ```
  private TraderServicePort trader_;
  ```

  The `TraderServicePort` object in the preceding line extends `weblogic.webservice.core.rpc.StubImpl`.

- Create a new instance of the `weblogic.webservice.core.rpc.StubImpl` object for each thread. This has some major performance (and coding) implications, so it should only be used as a last resort.

If your client application is not an EJB, you could also use synchronization to handle threading issues. You cannot use synchronization if your client is an EJB, because this would violate the EJB specifiation.

Because of the security and thread safety issues, the socket sharing mechanism is not enabled by default.

Enabling SSL Socket Sharing Using System Properties

To enable, using system properties, socket sharing in your SSL client application, set the Java system property `https.sharedsocket` to `true` on the command you use to invoke your client application, as shown in the following example:

```
java -Dbea.home=/bea_home \
     -Djava.protocol.handler.pkgs=com.certicom.net.ssl  \
     -Dhttps.sharedsocket=true my_app
```

The default value of the `https.sharedsocket` system property is `false`.

You can also specify the timeout value for shared sockets by using the `https.sharedsocket.timeout` system property to set the number of seconds that shared sockets live, as shown in the following example:

```
java -Dbea.home=/bea_home \
     -Djava.protocol.handler.pkgs=com.certicom.net.ssl  \
     -Dhttps.sharedsocket=true
     -Dhttps.sharedsocket.timeout=30 my_app
```

The default value of `https.sharedsocket.timeout` is 15 seconds.

Note: This timeout value does nothing to the actual transport layer controlling the socket. The value is used to determine if the SSL socket has not been referenced in the given timeframe and if not, then on this reference, if the time has expired, then the socket is closed and the protocol handshake is restarted.

Enabling SSL Socket Sharing Using the HttpsBindingInfo API

You can also use the `weblogic.webservice.binding.https.HttpsBindingInfo` SSL binding API, rather than system properties, to programmatically enable socket sharing from within your SSL client application. When you use the WebLogic SSL implementation, you use the public constructor of `HttpsBindingInfo` to create an `HttpsBindingInfo` object; the constructor specifies that the client application is using the `WLSSSLAdapter` subclass of the `SSLAdapter` class.

To enable socket sharing in your client application with the API, use the `HttpsBindingInfo.setSocketSharing(boolean)` setter method on the `HttpsBindingInfo` object, passing it a value of `true`. To disable socket sharting, pass the method a value of `false`. The default value, if you do not call this method in your application, is `false` (no socket sharing).

You can also specify the timeout value for shared sockets by using the `HttpsBindingInfo.setSharedSocketTimeout(long)` method on the `HttpsBindingInfo`

object, passing it the number of seconds that shared sockets live. The default value, if you do not set this method, is 15 seconds.

Note: This timeout value does nothing to the actual transport layer controlling the socket. The value is used to determine if the SSL socket has not been referenced in the given timeframe and if not, then on this reference, if the time has expired, then the socket is closed and the protocol handshake is restarted.

To close the shared SSL socket in your client application, use the `HttpsBindingInfo.closeSharedSocket()` method on the `HttpsBindingInfo` object. This method takes no parameters. Typically you close the shared socket in the cleanup method of the object from which you created the `HttpsBindingInfo` object.

Using a Third-Party SSL Implementation

If you want to use a third-party SSL implementation, you must first implement your own adapter class. The following example shows a simple class that provides support for JSSE; the main steps to implementing your own class are discussed after the example:

```
import java.net.URL;
import java.net.Socket;
import java.net.URLConnection;
import java.io.IOException;

public class JSSEAdapter implements weblogic.webservice.client.SSLAdapter {

  javax.net.SocketFactory factory =
    javax.net.ssl.SSLSocketFactory.getDefault();

  // implements weblogic.webservice.client.SSLAdapter interface...

  public Socket createSocket(String host, int port) throws IOException  {
    return factory.createSocket(host, port);
  }

  public URLConnection openConnection(URL url) throws IOException {
    // assumes you have java.protocol.handler.pkgs properly set..
    return url.openConnection();
  }

  // the configuration interface...

  public void setSocketFactory(javax.net.ssl.SSLSocketFactory factory) {
    this.factory = factory;
  }

  public javax.net.ssl.SSLSocketFactory getSocketFactory() {
    return (javax.net.ssl.SSLSocketFactory) factory;
```

```
  }
}
```

To create your own adapter class:

1. Create a class that implements the following interface:

   ```
   weblogic.webservice.client.SSLAdapter
   ```

2. Implement the `createSocket` method, whose signature is:

   ```
   public Socket createSocket(String host, int port)
       throws IOException
   ```

   This method returns an object that extends `java.net.Socket`. The object is connected to the designated hostname and port when a Web Service is invoked.

3. Implement the `openConnection` method, whose signature is:

   ```
   public URLConnection openConnection(URL url) throws IOException
   ```

   This method returns an object that extends the `java.net.URLConnection` class. The object is configured to connect to the designated URL. These connections are used for infrequent network operations, such as downloading the Web Service WSDL.

4. When you run your client application, set the following `System` property to the fully qualified name of your adapter class:

   ```
   weblogic.webservice.client.ssl.adapterclass
   ```

   The default `SSLAdapterFactory` class loads your adapter class and creates an instance of the class using the default no-argument constructor.

5. Configure your custom adapter class as shown in "Configuring the WebLogic SSL Implementation Programatically" on page 13-34, substituting your class for `WLSSLAdapter` and using the configuration methods defined for your adapter.

For detailed information, see the Web Service security Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/webservice/client/package-summary.html.

Extending the SSLAdapterFactory Class

You can create your own custom SSL adapter factory class by extending the `SSLAdapterFactory` class, which is used to create instances of adapters. One reason for extending the factory class is to allow custom configuration of each adapter when it is created, prior to use.

To create a custom SSL adapter factory class:

1. Create a class that extends the following class:

   `weblogic.webservice.client.SSLAdapterFactory`

2. Override the following method of the `SSLAdapterFactory` class:

   ```
   public weblogic.webservice.client.SSLAdapter createSSLAdapter();
   ```

   This method is called whenever a new `SSLAdapter`, or an adapter that implements this interface, is created by the adapter factory. By overriding this method, you can perform custom configuration of each new adapter before it is actually used.

3. In your client application, create an instance of your factory and set it as the default factory by executing the following method:

```
SSLAdapterFactory.setDefaultFactory(factoryInstance);
```

For detailed information, see the Web Service security Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/weblogic/webservice/client/package-summary.html.

Configuring Two-Way SSL For a Client Application

If you configured two-way SSL for WebLogic Server, the client application must present a certificate to WebLogic Server, in addition to WebLogic Server presenting a certificate to the client application as required by one-way SSL. The following sample Java code shows one way of doing this where the client application receives the client certificate file as an argument (relevant code in bold):

```
...

SSLAdapterFactory factory = SSLAdapterFactory.getDefaultFactory();
WLSSLAdapter adapter = (WLSSLAdapter) factory.getSSLAdapter();

if (argv.length > 1 ) {
  System.out.println("loading client certs from "+argv[1]);

FileInputStream clientCredentialFile  = new FileInputStream (argv[1]);
String pwd = "clientkey";

adapter.loadLocalIdentity(clientCredentialFile, pwd.toCharArray());

javax.security.cert.X509Certificate[] certChain = adapter.getIdentity("RSA",0);

factory.setDefaultAdapter(adapter);

factory.setUseDefaultAdapter(true);

...
```

Using a Proxy Server

If your client application is running inside a firewall, for example, and needs to use a proxy server, set the host name and the port of the proxy server using the following two System properties:

- **weblogic.webservice.transport.https.proxy.host**
- **weblogic.webservice.transport.https.proxy.port**

For more information on these System properties, see "Using Web Services System Properties" on page 7-14.

Configuring Access Control Security: Main Steps

Access control security refers to configuring the Web Service to control the users who are allowed to access it, and then coding your client application to authenticate itself, using HTTP, to the Web Service when the client invokes one of its operations.

The following procedure describes the high-level steps; later sections in the chapter describe the steps in more detail.

1. Control access to either the entire Web Service or some of its components by creating roles, mapping the roles to principals in your realm, then specifying which components are secured and accessible only by the principals in the role.

   See "Controlling Access to WebLogic Web Services" on page 13-41.

2. Optionally update the `web-services.xml` file to specify that the Web Service can be accessed *only* by HTTPS.

   See "Specifying the HTTPS Protocol" on page 13-44.

3. Code your client to authenticate itself using HTTP when invoking a WebLogic Web Service.

   See "Coding a Client Application to Authenticate Itself to a Web Service" on page 13-45.

Controlling Access to WebLogic Web Services

WebLogic Web Services are packaged as standard J2EE Enterprise applications. Consequently, to secure access to the Web Service, you secure access to some or all of the following components that make up the Web Service:

- The entire Web Service

- A subset of the operations of the Web Service
- The Web Service URL
- The stateless session EJB that implements the Web Service
- A subset of the methods of the stateless session EJB
- The WSDL and Home Page of the Web Service

You can use basic HTTP authentication or SSL to authenticate a client that is attempting to access a WebLogic Web Service. Because many of the preceding components are standard J2EE components, you secure them by using standard J2EE security procedures. The following sections describe how to secure each of these components.

Note: If the back-end component that implements your Web Service is a Java class or a JMS listener, the only way to secure the Web Service is by adding security constraints to the entire Web Service or to the URL that invokes the Web Service. In other words, you cannot secure just the back-end component that implements the Web Service.

For additional and detailed information about configuring, programming, and managing WebLogic security, see the security documentation at http://e-docs.bea.com/wls/docs81/security.html.

Securing the Entire Web Service and Its Operations

You secure an entire Web Service by creating a security policy through the Administration Console and assigning it to a WebLogic Web Service. You can also use the Administration Console to secure a subset of the Web Service operations. Security policies answer the question "who has access" to a WebLogic resource, in this case a Web Service or a subset of its operations. A security policy is created when you define an association between a WebLogic resource and a user, group, or role. A WebLogic resource has no protection until you assign it a security policy.

You assign security policies to an individual resource or to attributes or operations of a resource. If you assign a security policy to a type of resource, all new instances of that resource inherit that security policy. Security policies assigned to individual resources or attributes override security policies assigned to a type of resource.

To use a user or group to create a security policy, the user or group must be defined in the Authentication provider configured in the default security realm. To use a role to create a security policy, the role must be defined in the Role Mapping provider configured in the default security realm. By default, the WebLogic Authentication and Role Mapping providers are configured.

For more information and procedures about setting protections for a WebLogic Web Service or a subset of its operations using the Administration Console, see *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs81/secwlres/intro.html.

Securing the Web Service URL

Client applications use a URL to access a Web Service, as described in "WebLogic Web Services Home Page and WSDL URLs" on page 6-21. An example of such a URL is:

```
http://ariel:7001/web_services/TraderService
```

You can restrict access to the entire Web Service by restricting access to its URL. To do this, update the `web.xml` and `weblogic.xml` deployment descriptor files (in the Web application that contains the `web-services.xml` file) with security information.

For detailed information about restricting access to URLs, see *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs81/secwlres/index.html.

Securing the Stateless Session EJB and Its Methods

If you secure the stateless session EJB that implements a Web Service, client applications that invoke the service have access to the Web application, the WSDL, and the Web Service Home Page, but might not be able to invoke the actual method that implements an operation. This type of security is useful if you want to closely monitor who has access to the business logic of the EJB but do not want to block access to the entire Web Service.

You can also use this type of security to decide at the method-level who has access to the various operations of the Web Service. For example, you can specify that any user can invoke a method that views information, but only a certain subset of users are allowed to update the information.

For more information and procedures about securing EJBs and individual methods of an EJB using the Administration Console, see *Securing WebLogic Resources* at http://e-docs.bea.com/wls/docs81/secwlres/intro.html.

Securing the WSDL and Home Page of the Web Service

You can restrict access to either the WSDL or Home Page of a WebLogic Web Service by updating the `web-services.xml` deployment descriptor that describes the service, as described in the following procedure:

1. Open the `web-services.xml` file in your favorite editor.

The `web-services.xml` file is located in the `WEB-INF` directory of the Web application of the Web Services EAR file. See "The Web Service EAR File Package" on page 6-16 for more information on locating the file.

2. To restrict access to the WSDL, add the `exposeWSDL="False"` attribute to the `<web-service>` element that describes your Web Service. To restrict access to the Home page, add the `exposeHomePage="False"` attribute. The following excerpt shows an example:

```
<web-service
   name="stockquotes"
   uri="/myStockQuoteService"
   exposeWSDL="False"
   exposeHomePage="False" >
    ...
</web-service>
```

   The default value of the `exposeWSDL` and `exposeHomePage` attributes is `True`.

3. Re-deploy your Web Service for the change to take affect. The WSDL and Home Page of the Web Service will be inaccessible to all users.

Specifying the HTTPS Protocol

You make a Web Service accessible only through HTTPS by updating the `protocol` attribute of the `<web-service>` element in the `web-services.xml` file that describes the Web Service, as shown in the following excerpt:

```
<web-services>
  <web-service name="stockquotes"
               targetNamespace="http://example.com"
               uri="/myStockQuoteService"
               protocol="https" >
     ...
  </web-service>
</web-services>
```

Note: If you configure SSL for WebLogic Server and you do not specify the HTTPS protocol in the `web-services.xml` file, client applications can access the Web Service using *both* HTTP and HTTPS. However, if you specify HTTPS access in the `web-services.xml` file, client applications cannot use HTTP to access the Web Service.

If you use the `servicegen` Ant task to assemble the Web Service, use the `protocol` attribute of the `<service>` element to specify the HTTPS protocol, as shown in the following sample `build.xml` file:

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="ws_basic_statelessSession.ear"
       contextURI="WebServices"
       <service
         ejbJar="HelloWorldEJB.jar"
         targetNamespace="http://www.bea.com/webservices/basic/statelesSession"
         serviceName="HelloWorldEJB"
         serviceURI="/HelloWorldEJB"
         protocol="https"
         generateTypes="True"
         expandMethods="True">
       </service>
     </servicegen>
  </target>
</project>
```

Coding a Client Application to Authenticate Itself to a Web Service

When you write a JAX-RPC client application that invokes a Web Service, you use the following two properties to send a user name and password to the service so that the client can authenticate itself:

- `javax.xml.rpc.security.auth.username`
- `javax.xml.rpc.security.auth.password`

The following example, taken from the JAX-RPC specification, shows how to use these properties when using the `javax.xml.rpc.Stub` interfaces to invoke a secure Web Service:

```
StockQuoteProviderStub sqp = // ... get the Stub;
sqp._setProperty ("javax.xml.rpc.security.auth.username", "juliet");
sqp._setProperty ("javax.xml.rpc.security.auth.password", "mypassword");
float quote sqp.getLastTradePrice("BEAS");
```

If you use the WebLogic-generated client JAR file to invoke a Web Service, the Stub classes are already created for you, and you can pass the user name and password to the Service-specific implementation of the `getServicePort()` method, as shown in the following example taken from the JAX-RPC specification:

```
StockQuoteService sqs = // ... Get access to the service;
StockQuoteProvider sqp = sqs.getStockQuoteProviderPort ("juliet",
"mypassword");
float quote = sqp.getLastTradePrice ("BEAS");
```

In this example, the implementation of the `getStockQuoteProvidePort()` method sets the two authentication properties.

For additional information on writing a client application using JAX-RPC to invoke a secure Web Service, see http://java.sun.com/xml/jaxrpc/index.html.

Testing a Secure WebLogic Web Service From Its Home Page

The section "Deploying and Testing WebLogic Web Services" on page 6-21 describes how to invoke and test a *non-secure* WebLogic Web Service from its Home page.

Testing a secure WebLogic Web Service from its Home Page requires additional configuration of WebLogic Server, because in this case the server itself is acting as a secure client to the Web Service. In particular, you must configure WebLogic Server to use a trusted certificate authority (CA) file called `trusted-ca.pem`, which the Home Page is hard-coded to use when invoking the secure Web Service; the Home Page does *not* use the server keystore. This is because the Home Page uses the standard WebLogic Web Services client JAR file, which, with the aim of keeping the JAR file as thin as possible, does not include the security APIs needed to extract certificates from a keystore.

To test a secure WebLogic Web Service from its Home Page, follow these steps.

1. If not already configured, configure SSL for WebLogic Server.

   For more information, see *Configuring the SSL Protocol* at http://e-docs.bea.com/wls/docs81/secmanage/ssl.html.

2. Add the following flags to the script that starts up this instance of WebLogic Server:

```
-Dweblogic.webservice.client.ssl.strictcertchecking=false
-Dweblogic.security.SSL.ignoreHostnameVerification=true
```

3. Create a trusted certificate authority (CA) file called `trusted-ca.pem` by following these steps:

   a. Open a command window and set your environment.

      On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level

installation directory of the BEA products and *domainName* is the name of your domain.

On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*`/user_projects/domains/domainName`, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

b. Move to the *WL_HOME*`\server\lib` directory, where *WL_HOME* refers to the top-level WebLogic Platform installation directory.

c. Use the `utils.der2pem` WebLogic Server Java utility to convert the `WL_HOME\server\lib\CertGenCA.der` trusted certificate authority file from DER format to PEM:

```
java utils.der2pem CertGenCA.der
```

The utility generates a file called `CertGenCA.pem`.

d. Rename the generated `CertGenCA.pem` to `trusted-ca.pem`.

4. Move the `trusted-ca.pem` file you created in the preceding step to the domain directory of WebLogic Server.
5. Restart WebLogic Server for the startup flags to take effect.
6. Invoke the secure WebLogic Web Service's Home Page in your browser. The browser will return a message saying the certificate is not trusted.
7. Load the trusted certificate in your browser. You may need to restart your browser for it to take effect.
8. Invoke the secure WebLogic Web Service's Home Page again in your browser. You should now be able to test your secure Web Service as described in "Deploying and Testing WebLogic Web Services" on page 6-21.

CHAPTER 14

Internationalization

The Internationalization Guide at http://e-docs.bea.com/wls/docs81/i18n/index.html provides general information about internationalizing a WebLogic Server application. The following sections describe additional information specific to internationalizing a WebLogic Web Service:

- "Overview of Internationalization" on page 14-1
- "Internationalizing a WebLogic Web Service" on page 14-2
- "Invoking a Web Service Using a Specific Character Set" on page 14-4

Overview of Internationalization

Internationalization refers to the preparation of software so that it behaves properly in multiple locations. Internationalization of WebLogic Web Services primarily involves specifying the character set of the SOAP request and response. You then specify the character set of the SOAP request inside the client application that invokes the Web Service. There are a variety of ways to specify the character set that WebLogic Server uses in its SOAP response, as outlined in later sections. WebLogic Server can also accept many character sets in a SOAP request used to invoke a deployed WebLogic Web Service.

Often the default character sets used by WebLogic Server are adequate and you do not need to explicitly specify a character set for a Web Service. For example, if a client application specifies its preferred character set, and there is no character set specified for a Web Service, then WebLogic Server responds by using the client's preferred character set. Also, non-internationalized WebLogic Server instances use the `US-ASCII` character set by default, and internationalized WebLogic Server instances use the `UTF-8` character set by default, and both of

these character sets are compatible when one WebLogic Server instance is communicating with the other. This also means that a Web Service running on a non-internationalized WebLogic Server instance can handle multi-byte characters correctly.

However, if the default character sets are not adequate for your application, use the information here to specify the character set that you need.

Internationalizing a WebLogic Web Service

This section describes how to set the character set for a WebLogic Web Service. It also describes how WebLogic Server determines what character set it should use when sending the SOAP message response of an invoke of a deployed Web Service.

Specifying the Character Set for a WebLogic Web Service

When you specify the character set for a WebLogic Web Service, you are specifying the value of the `Content-Type` HTTP header of the SOAP message response to an invoke of a deployed Web Service. You use one of the following two methods to specify the character set for a WebLogic Web Service:

- Update the `web-services.xml` deployment descriptor file.
- Set the `weblogic.webservice.i18n.charset` WebLogic Server system property.

  Warning: This method specifies the character set for *all* deployed Web Services.

Updating the web-services.xml File

The preferred way to specify the character set used by a particular WebLogic Web Service is by updating its `web-serivces.xml` file.

To specify the character set for a WebLogic Web Service, update the `charset` attribute of the `<web-service>` element in the `web-services.xml` file. Set it equal to the standard name of the character set, as shown in the following sample excerpt:

```
<web-services>
     <web-service name="stockquotes"
                  targetNamespace="http://example.com"
                  uri="/myStockQuoteService"
                  charset="Shift_JIS">
      ...
     </web-service>
</web-services>
```

The default value is `US-ASCII`.

For the full list of character sets, see http://www.iana.org/assignments/character-sets.

If you set this attribute, the WebLogic Web Service *always* uses the specified character set in its SOAP response to an invoke of any operation in the Web Service.

Setting a WebLogic Server System Property

You can also specify the character set for *all* deployed WebLogic Web Services deployed on a WebLogic Server instance by setting the system property `weblogic.webservice.i18n.charset` equal to the name of the character set. Set this system property in the script that starts up the WebLogic Server instance:

```
-Dweblogic.webservice.i18n.charset=utf-8
```

Order of Precedence of Character Set Configuration Used By WebLogic Server

The following list shows the order by which WebLogic Server determines the character set of a WebLogic Web Service when it is creating the SOAP response to an invoke of one of its operations:

1. The value of the `charset` attribute in the corresponding `<web-service>` element of the `web-services.xml` deployment descriptor.

   If this is not set, then WebLogic Server looks at the following:

2. The character set preferred by the client application that invoked the Web Service operation. If your client application uses the WebLogic Web Services client APIS, the character set is specified using the `weblogic.webservice.binding.BindingInfo.setAcceptCharset()` method.

   If this is not set, then WebLogic Server looks at the following:

3. The value of the WebLogic Server system property `weblogic.webservice.i18n.charset`.

   If this is not set, then WebLogic Server looks at the following:

4. The character set specified for the JVM. Specifically, if the JVM property `user.language` is set to `en`, then WebLogic Web Services use the `US-ASCII` character set. If the `user.language` property is set to *anything* else, WebLogic Web Services use the `UTF-8` character set.

Invoking a Web Service Using a Specific Character Set

This section describes how to use WebLogic Web Service APIs to invoke a Web Service using a character set other than the default. The section also describes the character set settings in the HTTP request headers that are honored by WebLogic Web Services.

Setting the Character Set When Invoking a Web Service

If you use the WebLogic Web Service client APIs to invoke a Web Service, you use the `weblogic.webservice.binding.BindingInfo.setCharset()` to set the character set of the client application's SOAP request. In particular, this method sets the `Content-Type` HTTP header. This method sets the character set of *only* the data travelling from the client application to the Web Service. The SOAP response from the Web Service might use a completely different character set; see "Order of Precedence of Character Set Configuration Used By WebLogic Server" on page 14-3 for details on how to determine the character set of the SOAP response from a WebLogic Web Service.

Your client application can specify the character set that it would *prefer* the Web Service to use in its response by using the `weblogic.webservice.binding.BindingInfo.setAcceptCharset()` method. In particular, this method sets the `Accept-Charset` HTTP header.

The following code excerpt shows how to set the character set when invoking a Web Service operation, as well as specify the preferred character set in the response; in the example, `stub` is the instance of the JAX-RPC `Stub` class for your Web Service:

```
import weblogic.webservice.binding.BindingInfo;

...

    BindingInfo info =
     (BindingInfo)stub._getProperty("weblogic.webservice.bindinginfo" );

    // The following method sets the Content-Type HTTP header
    info.setCharset( "UTF-8" );
    port.helloWorld();

    // The following method sets the Accept-Charset HTTP header
    info.setAcceptCharset( "UTF-16" );
    port.helloWorld();
```

For more information about the `weblogic.webservice.binding` package, see the Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Warning: The `weblogic.webservice.binding` package is a proprietary WebLogic API; using it in your client applications might make it difficult to port them to non-WebLogic environments.

Character Set Settings in HTTP Request Headers Honored by WebLogic Web Services

When a WebLogic Web Service receives an HTTP SOAP request that invokes one of the service's operations, it honors HTTP headers as follows:

- WebLogic Web Services *always* honor the `charset` attribute of the `Content-Type` HTTP header, which specifies the character set of the SOAP request.
- WebLogic Web Services *sometimes* honor the `Accept-Charset` HTTP header. This header specifies the character set of the SOAP response preferred by the application that invoked the Web Service operation. If the WebLogic Web Service has not been configured with a specific character set (see "Specifying the Character Set for a WebLogic Web Service" on page 14-2), the SOAP response uses the character set specified by the `Accept-Charset` HTTP header. If, however, the WebLogic Web Service is configured to use a specific character set, that character set is *always* used in the SOAP response.
- WebLogic Web Services *never* honor the `encoding` attribute of the optional `<?xml?>` element that starts the SOAP 1.1 envelope.

  Note: This is true *only* for SOAP 1.1. For SOAP 1.2, if the `ContentType` HTTP Header is missing, then the `encoding` attribute of the `<?xml?>` element is honored.

The following excerpt of a SOAP envelope, including the HTTP headers, shows the three ways of specifying characters sets in bold:

```
POST /StockQuote HTTP/1.1
Host: www.sample.com
Content-Type: text/xml; charset="US-ASCII"
Content-Length: nnnn
SOAPAction: "Some-URI"
Accept-Charset: UTF-8

<?xml version="1.0" encoding="UTF-16"?>
<SOAP-ENV:Envelope
  ...
</SOAP-ENV:Envelope>
```

CHAPTER 15

Using SOAP 1.2

The following sections provide information about using SOAP 1.2 as the message format:

- "Overview of Using SOAP 1.2" on page 15-1
- "Specifying SOAP 1.2 for a WebLogic Web Service: Main Steps" on page 15-2
- "Updating the web-services.xml File Manually" on page 15-3
- "Invoking a Web Service Using SOAP 1.2" on page 15-3

Overview of Using SOAP 1.2

By default, a WebLogic Web Service uses SOAP 1.1 as the message format when a client application invokes one of its operations. You can, however, use SOAP 1.2 as the message format by updating the `web-services.xml` file and specifying a particular attribute in `clientgen` when you generate the client stubs.

Warning: BEA's SOAP 1.2 implementation is based on the W3C Working Draft specification (June 26, 2002). Because this specification is not yet a W3C Recommendation, BEA's current implementation is subject to change. BEA highly recommends that you use the SOAP 1.2 feature included in this version of WebLogic Server in a development environment *only*.

When a WebLogic Web Service is configured to use SOAP 1.2 as the message format:

- The generated WSDL of the Web Service contains *two* port definitions: one with a SOAP 1.1 binding, and another with a SOAP 1.2 binding.

- The `clientgen` Ant task, when generating the Web-service specific client JAR file for the Web Service, creates a `Service` implementation that contains *two* `getPort()` methods, one for SOAP 1.1 and another for SOAP 1.2.

Specifying SOAP 1.2 for a WebLogic Web Service: Main Steps

The following procedure assumes that you are familiar with the `servicegen` Ant task, and you want to update the Web Service to use SOAP 1.2 as the message format. For an example of using `servicegen`, see Chapter 3, "Creating a WebLogic Web Service: A Simple Example."

1. Update the `build.xml` file that contains the call to the `servicegen` Ant task, adding the attribute `useSOAP12="True"` to the `<service>` element that builds your Web Service, as shown in the following example:

```
<servicegen
   destEar="ears/myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="jars/myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True"
       useSOAP12="True" >
   </service>
</servicegen>
```

   Note: If you are not using `servicegen`, you can update the `web-services.xml` file of your WebLogic Web Service manually. For details, see "Updating the web-services.xml File Manually" on page 15-3.

2. Re-run the `servicegen` Ant task to regenerate your Web Service to use SOAP 1.2.

   For general details about the `servicegen` Ant task, see "Creating the Build File That Specifies the servicegen Ant Task" on page 6-4.

3. Re-run the `clientgen` Ant task.

Because the WSDL of the Web Service has been updated to include an additional port with a SOAP 1.2 binding, the `clientgen` Ant task automatically creates new stubs that contains these SOAP 1.2-specific `getPort()` methods.

For details, see "Generating the Client JAR File by Running the clientgen Ant Task" on page 7-5.

See "Invoking a Web Service Using SOAP 1.2" on page 15-3 for details about writing a Java client application that invokes your Web Service.

Updating the web-services.xml File Manually

The `web-services.xml` file is located in the `WEB-INF` directory of the Web application of the Web Services EAR file. See "The Web Service EAR File Package" on page 6-16 for more information on locating the file.

To update the `web-services.xml` file to specify SOAP 1.2:

1. Open the file in your favorite editor.
2. Add the `useSOAP12="True"` attribute to the `<web-service>` element that describes your Web Service. For example:

```
<web-service
   name="myWebService"
   useSOAP12="True"
   ...>
...
</web-service>
```

Invoking a Web Service Using SOAP 1.2

When writing your client application to invoke the SOAP 1.2-enabled WebLogic Web Service, you first use the `clientgen` Ant task to generate the Web Service-specific client JAR file that contains the generated stubs, as usual. The `clientgen` Ant task in this case generates a JAX-RPC `Service` implementation that contains *two* `getPort()` methods: the standard one for SOAP 1.1, called `get`*`ServiceName`*`Port()`, and a second one for SOAP 1.2, called `get`*`ServiceName`*`PortSoap12()`, where *`ServiceName`* refers to the name of your Web Service. These two `getPort()` methods correspond to the two port definitions in the generated WSDL of the Web Service, as described in "Overview of Using SOAP 1.2" on page 15-1.

The following example of a simple client application shows how to invoke the `helloWorld` operation of the `MyService` Web Service using both SOAP 1.1 (via the `getMyservicePort()` method) and SOAP 1.2 (via the `getMyServicePortSoap12()` method):

```
import java.io.IOException;

public class Main{

  public static void main( String[] args ) throws Exception{

    MyService service = new MyService_Impl();

    MyServicePort port = service.getMyServicePort();
    System.out.println( port.helloWorld() );

    port = service.getMyServicePortSoap12();
    System.out.println( port.helloWorld() );
  }
}
```

CHAPTER 16

Creating JMS-Implemented WebLogic Web Services

The following sections describe how to create JMS-implemented WebLogic Web Services:

- "Designing JMS-Implemented WebLogic Web Services" on page 16-2
- "Creating JMS-Implemented WebLogic Web Services" on page 16-3
- "Configuring JMS Components for Message-Style Web Services" on page 16-4
- "Assembling JMS-Implemented WebLogic Web Services Using servicegen" on page 16-5
- "Assembling JMS-Implemented WebLogic Web Services Manually" on page 16-7
- "Deploying JMS-Implemented WebLogic Web Services" on page 16-10
- "Invoking JMS-Implemented WebLogic Web Services" on page 16-10

Overview of JMS-Implemented WebLogic Web Services

In addition to implementing a Web Service operation with a stateless session EJB or a Java class, you can use a JMS message consumer or producer, such as a message-driven bean.

There are two types of JMS-implemented operations:

- Operations that send data to a JMS destination.

  You implement this type of operation with a JMS message consumer. The message consumer consumes the message after a client that invokes the Web Service operation sends data to the JMS destination.

- Operations that receive data from a JMS queue.

You implement this type of operation with a JMS message producer. The message producer puts a message on the specified JMS queue and a client invoking this message-style Web Service component polls and receives the message.

When a client application sends data to a JMS-implemented Web Service operation, WebLogic Server first converts the XML data to its Java representation using either the built-in or custom serializers, depending on whether the data type of the data is built-in or not. WebLogic Server then wraps the resulting Java object in a `javax.jms.ObjectMessage` object and puts it on the JMS destination. You can then write a JMS listener, such as a message-driven bean, to take the `ObjectMessage` and process it. Similar steps happen in reverse when a client application invokes a Web Service to receive data from a JMS queue.

If you are using non-built-in data types, you must update the `web-services.xml` deployment descriptor file with the correct data type mapping information. If the Web Service cannot find data type mapping information for the data, then it converts the data to a `javax.xml.soap.SOAPElement` data type, defined by the SOAP With Attachments API For Java (SAAJ) specification.

Note: Input and return parameters to a Web Service operation implemented with a JMS consumer or producer must implement the `java.io.Serializable` interface.

For detailed information about programming message-driven beans, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html.

Designing JMS-Implemented WebLogic Web Services

This section describes the relationship between JMS and WebLogic Web Services operations implemented with a JMS consumer or producer, and design considerations for developing these types of Web Services.

Retrieving and Processing Messages

After you decide what type of JMS destination you are going to use, you must decide what type of J2EE component will retrieve the message from the JMS destination and process it. Typically this will be a message-driven bean. This message-driven bean can do all the message-processing work, or it can parcel out some or all of the work to other EJBs. Once the message-driven bean finishes processing the message, the execution of the Web Service is complete.

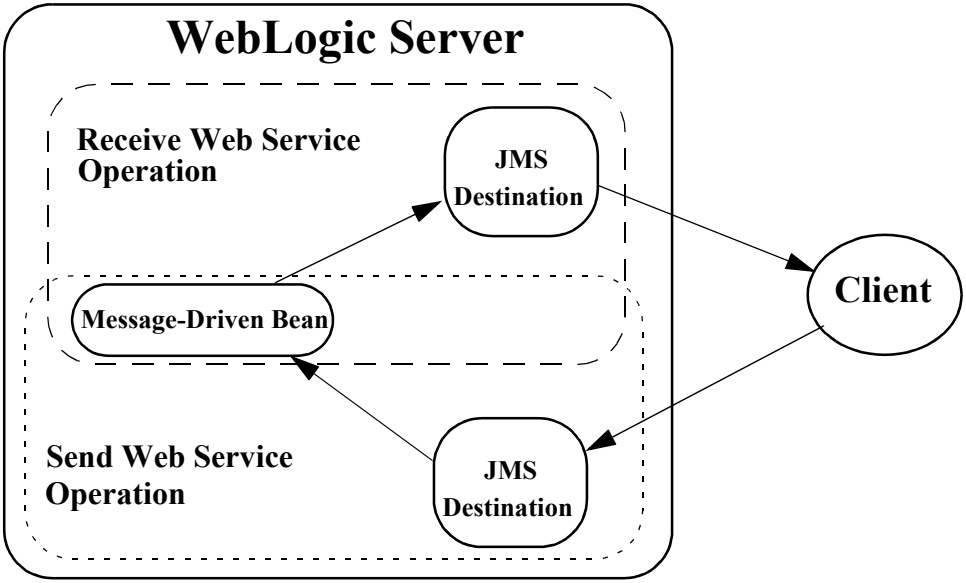
Because a single Web Service operation cannot both send and receive data, you must create two Web Service operations if you want a client application to be able to both send data and receive data. The sending Web Service operation is related to the receiving one because the original

message-driven bean that processed the message puts the response on the JMS destination corresponding to the receiving Web Service operation.

Example of Using JMS Components

Figure 16-1 shows two separate Web Service operations, one for receiving a message from a client and one for sending a message back to the client. The two Web Service operations have their own JMS destinations. The message-driven bean gets messages from the first JMS destination, processes the information, then puts a message back onto the second JMS destination. The client invokes the first Web Service operation to send the message to WebLogic Server and then invokes the second Web Service operation to receive a message back from WebLogic Server.

Figure 16-1 Data Flow Between JMS-Implemented Web Service Operations and JMS



Creating JMS-Implemented WebLogic Web Services

To create a Web Service implemented with a JMS message producer or consumer, follow these steps:

1. Write the Java code for the J2EE component (typically a message-driven bean) that will consume or produce the message from or on the JMS destination.

For detailed information, see *Programming WebLogic Enterprise JavaBeans* at http://e-docs.bea.com/wls/docs81/ejb/index.html.

2. Use the Administration Console to configure the following JMS components of WebLogic Server:
   - The JMS queue that will either receive the XML data from a client or send XML data to a client. Later, when you assemble the Web Service as described in Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks," you will use the name of this JMS destination.
   - The JMS Connection factory that the WebLogic Web Service uses to create JMS connections.

   For more information on this step, see "Configuring JMS Components for Message-Style Web Services" on page 16-4.

Configuring JMS Components for Message-Style Web Services

In this section it is assumed that you have already configured a JMS server. For information about configuring JMS servers, and general information about JMS, see *JMS: Configuring* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html and *Programming WebLogic JMS* at http://e-docs.bea.com/wls/docs81/jms/index.html.

To configure a JMS queue and JMS Connection Factory, follow these steps:

1. Invoke the Administration Console in your browser. For details, see "Overview of Administering WebLogic Web Services" on page 17-1.
2. In the left pane, open Services—JMS.
3. Right-click the Connection Factories node and choose Configure a new JMSConnectionFactory from the drop-down list.
4. Enter a name for the Connection Factory in the Name field.
5. Enter the JNDI name of the Connection Factory in the JNDIName field.
6. Enter values in the remaining fields as appropriate. For information on these fields, see *JMS: Configuring* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html.
7. Click Create.
8. Select the servers or clusters on which you would like to deploy this JMS connection factory.

9. Click Apply.
10. In the left pane, open Services—JMS—Servers.
11. Select the JMS server for which you want to create a JMS destination.
12. Right-click the Destinations node and choose from the drop-down list Configure a new JMSQueue to create a queue.
13. Enter the name of the JMS destination in the Name text field.
14. Enter the JNDI name of the destination in the JNDIName text field.
15. Enter values in the remaining fields as appropriate. For information on these fields, see *JMS: Configuring* at http://e-docs.bea.com/wls/docs81/ConsoleHelp/jms_config.html.
16. Click Create.

Assembling JMS-Implemented WebLogic Web Services Using servicegen

You can use the `servicegen` Ant task to assemble a JMS-implemented Web Service automatically. The Ant task creates a `web-services.xml` deployment descriptor file based on the attributes of the `build.xml` file, optionally creates the non-built-in data type components (such as serialization class), optionally creates a client JAR file used to invoke the Web Service, and packages everything into a deployable EAR file.

To automatically assemble a Web Service using the `servicegen` Ant task:

1. Create a staging directory to hold the components of your Web Service.
2. Package your JMS message consumers and producers (such as message-driven beans) into a JAR file.

   For detailed information on this step, refer to *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/programming/environment.html.
3. Copy the JAR file to the staging directory.
4. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`/user_projects/domains/domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

5. In the staging directory, create the Ant build file (called `build.xml` by default) that contains a call to the `servicegen` Ant task.

   For details about specifying the `servicegen` Ant task, see Listing 16-1.

   For general information about creating Ant build files, see http://jakarta.apache.org/ant/manual/.

6. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the staging directory, optionally passing the command a target argument:

   ```
   prompt> ant
   ```

   The Ant task generates the Web Services EAR file in the staging directory which you can then deploy on WebLogic Server.

The following sample `build.xml` file contains a call to the `servicegen` Ant task.

Listing 16-1 Sample build.xml File

```
<project name="buildWebservice" default="ear">
  <target name="ear">
    <servicegen
       destEar="jms_send_queue.ear"
       contextURI="WebServices" >
       <service
           JMSDestination="jms.destination.queue1"
           JMSAction="send"
           JMSDestinationType="queue"
           JMSConnectionFactory="jms.connectionFactory.queue"
           JMSOperationName="sendName"
           JMSMessageType="types.myType"
           generateTypes="True"
           targetNamespace="http://tempuri.org"
           serviceName="jmsSendQueueService"
           serviceURI="/jmsSendQueue"
           expandMethods="True">
```

```
      </service>
    </servicegen>
  </target>
</project>
```

When you run the `servicegen` Ant task using the preceding `build.xml` file, the Ant task creates a single Web Service called `jmsSendQueueService`. The URI to identify this Web Service is `/jmsSendQueue`; the full URL to access the Web Service is

```
http://host:port/WebServices/jmsSendQueue
```

The `servicegen` Ant task packages the Web Service in an EAR file called `jms_send_queue.ear`. The EAR file contains a WAR file called `web-services.war` (default name) that contains all the Web Service components, such as the `web-services.xml` deployment descriptor file.

The Web Service exposes a single operation called `sendName`. Client applications that invoke this Web Service operation send messages to a JMS queue whose JNDI name is `jms.destination.queue1`. The JMS `ConnectionFactory` used to create the connection to this queue is `jms.connectionFactory.queue`. The data type of the single parameter of the `sendName` operation is `types.myType`. Because the `generateTypes` attribute is set to `True`, the `servicegen` Ant task generates the non-built-in data type components for this data type, such as the serialization class.

Note: The `types.myType` Java class must be in `servicegen`'s CLASSPATH so that `servicegen` can generate appropriate components.

Assembling JMS-Implemented WebLogic Web Services Manually

To assemble a JMS-implemented WebLogic Web Service manually:

1. Package the JMS message consumers and producers into a JAR file. See "Packaging the JMS Message Consumers and Producers" on page 16-8.
2. Update the `web-services.xml` file with component information. See "Updating the web-services.xml File With Component Information" on page 16-8.
3. Follow the steps described in "Assembling WebLogic Web Services Using Individual Ant Tasks" on page 6-5, using the JMS-specific information where appropriate.

The following sections describe JMS-specific information about assembling Web Services manually.

Packaging the JMS Message Consumers and Producers

Package your JMS message consumers and producers (such as message-driven beans) into a JAR file.

When you create the EAR file that contains the entire Web Service, put this JAR file in the top-level directory, in the same location as EJB JAR files.

Updating the web-services.xml File With Component Information

Use the `<components>` child element of the `<web-service>` element to list and describe the JMS back-end components that implement the operations of the Web Service. Each back-end component has a `name` attribute that you later use when describing the operation that the component implements.

See "Sample web-services.xml File for JMS Component Web Service" on page 16-9 for an example.

You list the following types of back-end components for JMS-implemented Web Services:

- `<jms-send-destination>`

  This element describes a JMS back-end component to which client applications send data. The component puts the sent data on to a JMS destination. Use the `connection-factory` attribute of this element to specify the JMS Connection factory that WebLogic Server uses to create a JMS Connection object. Use the `<jndi-name>` child element to specify the JNDI name of the destination, as shown in the following example:

  ```
  <components>
     <jms-send-destination name="inqueue"
                           connection-factory="myapp.myqueueCF">
         <jndi-name path="myapp.myqueueIN" />
     </jms-send-destination>
  </components>
  ```

- `<jms-receive-queue>`

  This element describes the JMS back-end component in which client applications receive data, in particular from a JMS queue. Use the `connection-factory` attribute to specify the JMS Connection factory that WebLogic Server users to create a JMS Connection

object. Use the `<jndi-name>` child element to specify the JNDI name of the queue, as shown in the following example:

```
<components>
   <jms-receive-queue name="outqueue"
                      connection-factory="myapp.myqueueCF">
       <jndi-name path="myapp.myqueueOUT" />
   </jms-receive-queue>
</components>
```

Sample web-services.xml File for JMS Component Web Service

The following sample `web-services.xml` file describes a Web Service that is implemented with a JMS message consumer or producer:

```
<web-services>
     <web-service targetNamespace="http://example.com"
                  name="myMessageService" uri="MessageService">

         <components>
            <jms-send-destination name="inqueue"
                                  connection-factory="myapp.myqueuecf">
                 <jndi-name path="myapp.myinputqueue" />
            </jms-send-destination>
            <jms-receive-queue name="outqueue"
                               connection-factory="myapp.myqueuecf">
                 <jndi-name path="myapp.myoutputqueue" />
            </jms-receive-queue>
         </components>

         <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
              <operation invocation-style="one-way" name="enqueue"
                         component="inqueue" />
                 <params>
                   <param name="input_payload" style="in" type="xsd:anyType" />
                 </params>
              </operation>
              <operation invocation-style="request-response" name="dequeue"
                         component="outqueue" >
                 <params>
                   <return-param name="output_payload" type="xsd:anyType"/>
                 </params>
              </operation>
         </operations>
     </web-service>
</web-services>
```

The example shows two JMS back-end components, one called `inqueue` in which a client application sends data to a JMS destination, and one called `outqueue` in which a client application receives data from a JMS queue.

Two corresponding Web Service operations, `enqueue` and `dequeue`, are implemented with these back-end components.

The `enqueue` operation is implemented with the `inqueue` component. This operation is defined to be asynchronous one-way, which means that the client application, after sending the data to the JMS destination, does not receive a SOAP response (not even an exception.) The data sent by the client is contained in the `input_payload` parameter.

The `dequeue` operation is implemented with the `outqueue` component. The `dequeue` operation is defined as synchronous request-response because the client application invokes the operation to receive data from the JMS queue. The response data is contained in the output parameter `output_payload`.

Deploying JMS-Implemented WebLogic Web Services

Deploying a WebLogic Web Service refers to making it available to remote clients. Because WebLogic Web Services are packaged as standard J2EE Enterprise applications, deploying a Web Service is the same as deploying an Enterprise application.

For detailed information on deploying Enterprise applications, see *Deploying WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/deployment/index.html.

Invoking JMS-Implemented WebLogic Web Services

This section shows two sample client applications that invoke JMS-implemented WebLogic Web Services: a client application that sends data to a service operation, and a client application that receives data from another operation within the same Web Service. The first operation is implemented with a JMS destination, the second with a JMS queue, as shown in the following `web-services.xml` file that describes the Web Service:

```
<web-services xmlns:xsd="http://www.w3.org/2001/XMLSchema" >

  <web-service
    name="BounceService"
    targetNamespace="http://www.foobar.com/echo"
    uri="/BounceService">

      <components>
```

```
        <jms-send-destination name="inqueue"
                      connection-factory="weblogic.jms.ConnectionFactory">
             <jndi-name path="weblogic.jms.inqueue" />
        </jms-send-destination>
        <jms-receive-queue name="outqueue"
                      connection-factory="weblogic.jms.ConnectionFactory">
            <jndi-name path="weblogic.jms.outqueue" />
        </jms-receive-queue>
   </components>

   <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
      <operation invocation-style="one-way" name="submit" component="inqueue" >
      </operation>

      <operation invocation-style="request-response"
                 name="query" component="outqueue" >
         <params>
             <return-param name="output_payload" type="xsd:string"/>
        </params>
      </operation>
   </operations>

 </web-service>

</web-services>
```

Invoking an Asynchronous Web Service Operation to Send Data

The following example shows a dynamic client application that invokes the `submit` operation, described in the `web-services.xml` file in the preceding section. The `submit` operation sends data from the client application to the `weblogic.jms.inqueue` JMS destination. Because the operation is defined as `one-way`, it is asynchronous and does not return any value to the client application that invoked it.

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */
```

```
/**
 * send2WS - this module sends to a specific Web Service connected JMS queue
 * If the message is 'quit' then the module exits.
 *
 * @returns
 * @throws Exception
 */

public class send2WS{

  public static void main( String[] args ) throws Exception {

  // Setup the global JAX-RPC service factory
  System.setProperty( "javax.xml.rpc.ServiceFactory",
    "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  ServiceFactory factory = ServiceFactory.newInstance();

  //define qnames
  String targetNamespace = "http://www.foobar.com/echo";

  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );

  //create service
  Service service = factory.createService( serviceName );

  //create outbound call
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "submit" );

  //set port and operation name
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //add parameters
  Ws2JmsCall.addParameter( "param",
        new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
ParameterMode.IN

);
  //set end point address
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  // get message from user
  BufferedReader msgStream =
      new BufferedReader(new InputStreamReader(System.in));
  String line = null;
  boolean quit = false;
```

```
  while (!quit) {
    System.out.print("Enter message (\"quit\" to quit): ");
    line = msgStream.readLine();
    if (line != null && line.trim().length() != 0) {
          String result = (String)Ws2JmsCall.invoke( new Object[]{ line } );
      if(line.equalsIgnoreCase("quit")) {
            quit = true;
            System.out.print("Done!");
        }
      }
    }
  }
}
```

Invoking a Synchronous Web Service Operation to Send Data

The following example shows a dynamic client application that invokes the `query` operation, described in the `web-services.xml` file in "Invoking JMS-Implemented WebLogic Web Services" on page 16-10. The client application invoking the `query` operation receives data from the `weblogic.jms.outqueue` JMS queue. Because the operation is defined as `request-response`, it is synchronous and returns the data from the JMS queue to the client application.

```
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.Service;
import javax.xml.rpc.Call;
import javax.xml.rpc.ParameterMode;

import javax.xml.namespace.QName;

/**
 * @author Copyright (c) 2002 by BEA Systems, Inc. All Rights Reserved.
 */

/**
 * fromWS - this module receives from a Web Service associated JMS queue
 * If the message is 'quit' then the module exits.
 *
 * @returns
 * @throws Exception
 */

public class fromWS {

  public static void main( String[] args ) throws Exception {

  boolean quit = false;
  // Setup the global JAX-RPC service factory
```

```
  System.setProperty( "javax.xml.rpc.ServiceFactory",
      "weblogic.webservice.core.rpc.ServiceFactoryImpl");

  ServiceFactory factory = ServiceFactory.newInstance();

  //define qnames
  String targetNamespace = "http://www.foobar.com/echo";

  QName serviceName = new QName( targetNamespace, "BounceService" );
  QName portName = new QName( targetNamespace, "BounceServicePort" );

  //create service
  Service service = factory.createService( serviceName );

  //create outbound call
  Call Ws2JmsCall = service.createCall();

  QName operationName = new QName( targetNamespace, "query" );

  //set port and operation name
  Ws2JmsCall.setPortTypeName( portName );
  Ws2JmsCall.setOperationName( operationName );

  //add parameters
  Ws2JmsCall.addParameter( "output_payload",
        new QName( "http://www.w3.org/2001/XMLSchema", "string" ),
                    ParameterMode.OUT );
  //set end point address
  Ws2JmsCall.setTargetEndpointAddress(
        "http://localhost:7001/BounceBean/BounceService" );

  System.out.println("Setup complete.  Waiting for a message...");

  while (!quit) {
        String result = (String)Ws2JmsCall.invoke( new Object[] {} );
        if(result != null) {
          System.out.println("TextMessage:" + result);
      if (result.equalsIgnoreCase("quit")) {
            quit = true;
        System.out.println("Done!");
          }
        continue;
        }
    try {Thread.sleep(2000);} catch (Exception ignore) {}
  }
 }
}
```

CHAPTER 17

Administering WebLogic Web Services

The following sections describe tasks for administering WebLogic Web Services:

- "Overview of Administering WebLogic Web Services" on page 17-1
- "Using the Administration Console to Administer Web Services" on page 17-2

Overview of Administering WebLogic Web Services

Once you develop and assemble a WebLogic Web Service, you can use the Administration Console to deploy it on WebLogic Server. Additionally, you can use the Administration Console to perform standard WebLogic administration tasks on the deployed Web Services, such as undeploy, delete, view, and so on.

Typically, a Web Service is packaged as an EAR file. The EAR file consists of a WAR file that contains the `web-services.xml` file and optional Java classes (such as the Java classes that implement a Web Service, handlers, and serialization classes for non-built-in data types) and a optional EJB JAR files that contain the stateless EJBs that implement the Web Service operations. The `servicegen` Ant task always packages a Web Service into an EAR file.

You can also package a Web Service as just a Web application WAR file if the operations are implemented with only Java classes, and not EJBs.

The Administration Console identifies a Web Service by the contents of the WAR file. In other words, if the WAR file contained in an EAR file contains a `web-services.xml` file, then the Administration Console lists the WAR file as a Web Service. The Administration Console uses the icon to indicate that the WAR file is in fact a Web Service.

To invoke the Administration Console in your browser, enter the following URL:

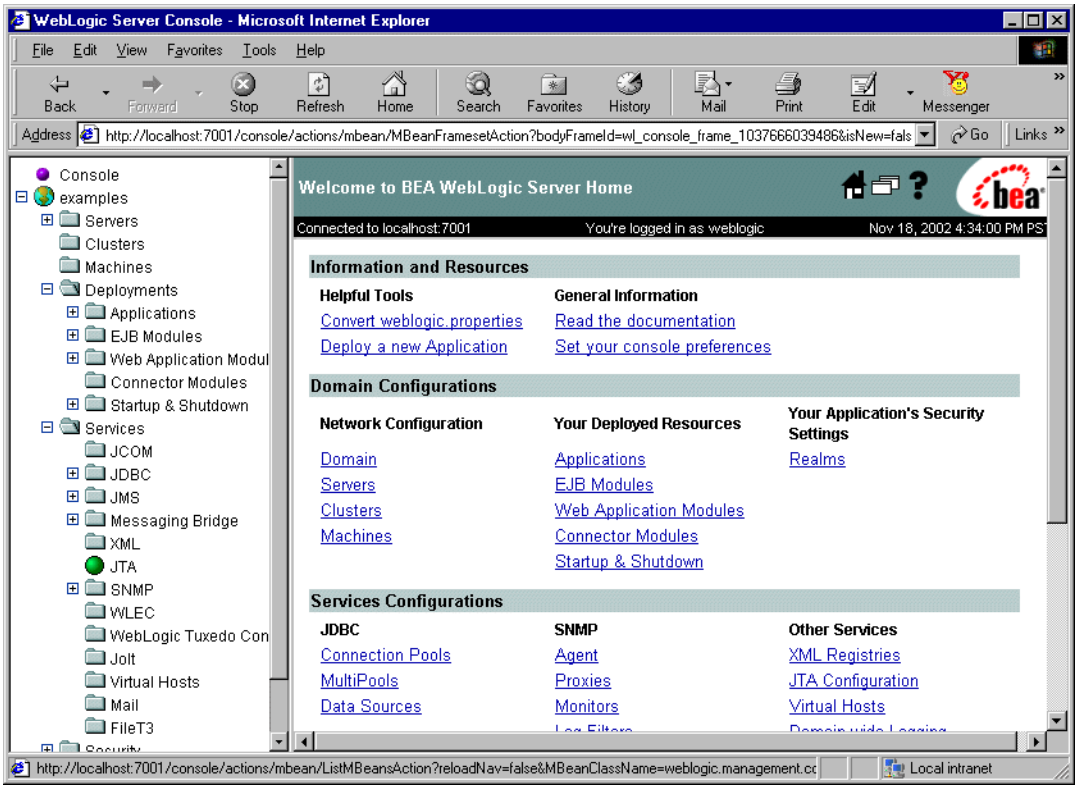
```
http://host:port/console
```

where

- *host* refers to the computer on which the Administration Server is running.
- *port* refers to the port number where the Administration Server is listening for connection requests. The default port number for the Administration server is 7001.

The following figure shows the main Administration Console window.

Figure 17-1 WebLogic Server Administration Console Main Window



Using the Administration Console to Administer Web Services

You can perform the following tasks using the Administration Console:

- Configure and Deploy a New Web Service at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#deploy_ws
- View a Deployed Web Service at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#view_ws
- Undeploy a Deployed Web Service at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#undeploy_ws
- Delete a Web Service at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#delete_ws
- View the Web Service Deployment Descriptor at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#view_dd_ws
- Configure Reliable SOAP Messaging at
  http://e-docs.bea.com/wls/docs81/ConsoleHelp/webservices.html#reliable_messaging

CHAPTER 18

Publishing and Finding Web Services Using UDDI

The following sections provide information about publishing and finding Web Services using UDDI:

- "Overview of UDDI" on page 18-1
- "WebLogic Server UDDI Features" on page 18-4
- "UDDI 2.0 Server" on page 18-5
- "UDDI Directory Explorer" on page 18-15
- "UDDI Client API" on page 18-16
- "Pluggable tModel" on page 18-16

Overview of UDDI

UDDI stands for Universal Description, Discovery and Integration. The UDDI Project is an industry initiative that is working to enable businesses to quickly, easily, and dynamically find and carry out transactions with one another.

A populated UDDI registry contains cataloged information about businesses, the services that they offer and communication standards and interfaces they use to conduct transactions.

Built on the Simple Object Access Protocol (SOAP) data communication standard, UDDI creates a global, platform-independent, open architecture space that will benefit businesses.

The UDDI registry can be broadly divided into two categories:

- UDDI and Web Services
- UDDI and Business Registry

For details about the UDDI data structure, see "UDDI Data Structure" on page 18-3.

UDDI and Web Services

The owners of Web Services publish them to the UDDI registry. Once published, the UDDI registry maintains pointers to the Web Service description and to the service.

The UDDI allows clients to search this registry, find the intended service and retrieve its details. These details include the service invocation point as well as other information to help identify the service and its functionality.

Web Service capabilities are exposed through a programming interface, and usually explained through Web Services Description Language (WSDL). In a typical publish-and-inquire scenario, the provider publishes its business, registers a service under it and defines a binding template with technical information on its Web Service. The binding template also holds reference to one or several *tModels*, which represent abstract interfaces implemented by this Web Service. The tModels might have been uniquely published by the provider, with information on the interfaces and URL references to the WSDL document.

A typical client inquiry may have one of two objectives:

1. To seek an implementation of a known interface.

   In other words, the client has a tModel ID and seeks binding templates referencing that tModel.

2. To seek the updated value of the invocation point (i.e., access point) of a known binding template ID.

UDDI and Business Registry

As a Business Registry solution, UDDI enables companies to advertise the business products and services they provide, as well as how they conduct business transactions on the Web. This use of UDDI has the potential of fueling growth of business-to-business (B2B) electronic commerce.

The minimum required information to publish a business is a single business name. Once completed, a full description of a business entity may contain a wealth of information, all of which helps to advertise the business entity and its products and services in a precise and accessible manner.

A Business Registry may contain the following:

- **Business Identification**—Multiple names and descriptions of the business, comprehensive contact information and standard business identifiers such as a tax identifier.
- **Categories**—Standard categorization information (for example a D-U-N-S business category number).
- **Service Description**—Multiple names and descriptions of a service. As a container for service information, companies can advertise numerous services, while clearly displaying the ownership of services. The `bindingTemplate` information describes how to access the service.
- **Standards Compliance**—In some cases it is important to specify compliance with standards. These standards might display detailed technical requirements on how to use the service.
- **Custom Categories**—It is possible to publish proprietary specifications (tModels) that identify or categorize businesses or services.

UDDI Data Structure

The data structure within UDDI is comprised of four constructions: a `businessEntity` structure, a `businessService` structure, a `bindingTemplate` structure and a `tModel` structure.

The following table outlines the difference between these constructions when used for Web Service or Business Registry applications.

Table 18-1 UDDI Data Structure

Data Structure	Web Service	Business Registry
businessEntity	Represents a Web Service provider: • Company name • Contact detail • Other business information	Represents a company, a division or a department within a company: • Company name(s) • Contact details • Identifiers and Categories
businessService	A logical group of one or several Web Services. API(s) with a single name stored as a child element, contained by the business entity named above.	A group of services may reside in a single `businessEntity`. • Multiple names and descriptions • Categories • Indicators of compliancy with standards
bindingTemplate	A single Web Service. Information provided here gives client applications the technical information needed to bind and interact with the target Web Service. Contains access point (i.e., URI to invoke a Web Service).	Further instances of standards conformity. Access points for the service in form of URLs, phone numbers, email addresses, fax numbers or other similar address types.
tModel	Represents a technical specification; typically a specifications pointer, or metadata about a specification document, including a name and a URL pointing to the actual specifications. In the context of Web Services, the actual specifications document is presented in the form of a WSDL file.	Represents a standard or technical specification, either well established or registered by a user for specific use.

WebLogic Server UDDI Features

Weblogic Server provides the following UDDI features:

- UDDI 2.0 Server

- UDDI Directory Explorer
- UDDI Client API.
- Pluggable tModel

UDDI 2.0 Server

The UDDI 2.0 Server is part of WebLogic Server and is started automatically when WebLogic Server is started. The UDDI Server implements the UDDI 2.0 server specification at http://www.uddi.org/specification.html.

Configuring the UDDI 2.0 Server

To configure the UDDI 2.0 Server:

1. Stop WebLogic Server.
2. Update the `uddi.properties` file, located in the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Platform installation directory.

   Warning: If your WebLogic Server domain was created by a user different from the user that installed WebLogic Server, the WebLogic Server administrator must change the permissions on the `uddi.properties` file to give access to all users.
3. Restart WebLogic Server.

Never edit the `uddi.properties` file while WebLogic Server is running. Should you modify this file in a way that prevents the successful startup of UDDI Server, refer to the *`WL_HOME`*`/server/lib/uddi.properties.booted` file for the last known good configuration.

To restore your configuration to its default, remove the `uddi.properties` file from the *`WL_HOME`*`/server/lib` directory. BEA strongly recommends that you move this file to a backup location, because a new `uddi.properties` file will be created and with its successful startup the `uddi.properties.booted` file will also be overwritten. After removing the properties file, start the server. Minimal default properties will be loaded and written to a newly created `uddi.properties` file.

The following section describes the UDDI Server properties that you can include in the `uddi.properites` file. The list of properties has been divided according to component, usage and functionality. At any given time, you do not need all these properties to be present.

Description of Properties in the uddi.properties File

The following tables describe all the properties of the `uddi.properties` file, categorized by the type of UDDI feature they configure:

- Basic UDDI Configuration
- UDDI User Defaults
- General Server Configuration
- Logger Configuration
- Connection Pools
- LDAP Datastore Configuration
- Replicated LDAP Datastore Configuration
- File Datastore Configuration
- General Security Configuration
- LDAP Security Configuration
- File Security Configuration

Table 18-2 Basic UDDI Configuration

UDDI Property Key	Description
auddi.discoveryurl	Specifies the DiscoveryURL prefix that is set for each saved business entity. This will typically be the full URL to the uddilistener servlet, so that the full DiscoveryURL results in the display of the stored BusinessEntity data.
auddi.inquiry.secure	Permissible values are `true` and `false`. When set to `true`, inquiry calls to UDDI Server will be limited to secure https connections only. Any UDDI inquiry calls through a regular http URL will be rejected.
auddi.publish.secure	Permissible values are `true` and `false`. When set to `true`, publish calls to UDDI Server will be limited to secure https connections only. Any UDDI publish calls through a regular http URL will be rejected.

Table 18-2 Basic UDDI Configuration

UDDI Property Key	Description
auddi.search.maxrows	The value of this property specifies the maximum number of returned rows for search operations. When the search results in a higher number of rows then the limit set by this property, the result will be truncated.
auddi.search.timeout	The value of this property specifies a timeout value for search operations. The value is indicated in milliseconds.
auddi.siteoperator	This property determines the name of the UDDI registry site operator. The specified value will be used as the operator attribute, saved in all future BusinessEntity registrations. This attribute will later be returned in responses, and indicates which UDDI registry has generated the response.
security.cred.life	The value of this property, in seconds, specifies the credential life for authentication. Upon authentication of a user, an AuthToken is assigned which will be valid for the duration specified by this property.
pluggableTModel.file.list	UDDI Server is pre-populated with a set of Standard TModels. You can further customize the UDDI server by providing your own taxonomies, in the form of TModels. Taxonomies must be defined in XML files, following the provided XML schema. The value of this property a comma-separated list of URIs to such XML files. Values that refer to these TModels will be checked and validated against the specified taxonomy.

Table 18-3 UDDI User Defaults

UDDI Property Key	Description
auddi.default.lang	The value of this property determines a user's initial language, assigned to his user profile by default at the time of creation. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.assertion	The value of this property determines a user's initial assertion quota, assigned to his user profile by default at the time of creation. The assertion quota is the maximum number of publisher assertions that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.binding	The value of this property determines a user's initial binding quota, assigned to his user profile by default at the time of creation. The binding quota is the maximum number of binding templates that the user is allowed to publish, per each business service. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.entity	The value of this property determines a user's initial business entity quota, assigned to his user profile by default at the time of creation. The entity quota is the maximum number of business entities that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.messageSize	The value of this property determines a user's initial message size limit, assigned to his user profile by default at the time of creation. The message size limit is the maximum size of a SOAP call that the user may send to UDDI Server. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.

Table 18-3 UDDI User Defaults

UDDI Property Key	Description
auddi.default.quota.service	The value of this property determines a user's initial service quota, assigned to his user profile by default at the time of creation. The service quota is the maximum number of business services that the user is allowed to publish, per each business entity. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.
auddi.default.quota.tmodel	The value of this property determines a user's initial TModel quota, assigned to his user profile by default at the time of creation. The TModel quota is the maximum number of TModels that the user is allowed to publish. To not impose any limits, set a value of -1 for this property. A user's profile settings may be changed either at sign-up or later.

Table 18-4 General Server Configuration

UDDI Property Keys	Description
auddi.datasource.type	This property allows you to configure the physical storage of UDDI data. This value defaults to `WLS`. The value of `WLS` for this property indicates that the internal LDAP directory of WebLogic Server is to be used for data storage. Other permissible values include `LDAP`, `ReplicaLDAP`, and `File`.
auddi.security.type	This property allows you to configure UDDI Server's security module (authentication). This value defaults to `WLS`. The value of `WLS` for this property indicates that the default security realm of WebLogic Server is to be used for UDDI authentication. As such, a WebLogic Server user would be an UDDI Server user and any WebLogic Server administrator would also be an UDDI Server administrator, in addition to members of the UDDI Server administrator group, as defined in UDDI Server settings. Other permissible values include `LDAP` and `File`.

Table 18-4 General Server Configuration

UDDI Property Keys	Description
auddi.license.dir	The value of this property specifies the location of the UDDI Server license file. In the absence of this property, the *WL_HOME*/server/lib directory is assumed to be the default license directory, where *WL_HOME* is the main WebLogic Platform installation directory. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).
auddi.license.file	The value of this property specifies the name of the license file. In the absence of this property, uddilicense.xml is presumed to be the default license filename. Some WebLogic users are exempt from requiring an UDDI Server license for the basic UDDI Server components, while they may need a license for additional components (e.g., UDDI Server Browser).

Table 18-5 Logger Configuration

UDDI Property Key	Description
logger.file.maxsize	The value of this property specifies the maximum size of logger output files (if output is sent to file), in Kilobytes. Once an output file reaches maximum size, it is closed and a new log file is created.
logger.indent.enabled	Permissible values are true and false. When set to true, log messages beginning with "+" and "-", typically TRACE level logs, cause an increase or decrease of indentation in the output.
logger.indent.size	The value of this property, an integer, specifies the size of each indentation (how many spaces for each indent).
logger.log.dir	The value of this property specifies an absolute or relative path to a directory where log files are stored.
logger.log.file.stem	The value of this property specifies a string that is prefixed to all log file names.

Table 18-5 Logger Configuration

UDDI Property Key	Description
logger.log.type	The value of this property determines whether log messages are sent to the screen, to a file or to both destinations. Permissible values for this property, respectively are: `LOG_TYPE_SCREEN`, `LOG_TYPE_FILE`, and `LOG_TYPE_SCREEN_FILE`.
logger.output.style	The value of this property determines whether logged output will simply contain the message, or if thread and timestamp information will be included. The two permissible values are `OUTPUT_LONG` and `OUTPUT_SHORT`.
logger.quiet	The value of this property determines whether the logger itself displays information messages or not. Permissible values are `true` and `false`.
logger.verbosity	The value of this property determines the logger's verbosity level. Permissible values (case sensitive) are `TRACE`, `DEBUG`, `INFO`, `WARNING` and `ERROR`, where each severity level includes the following ones accumulatively.

Table 18-6 Connection Pools

UDDI Property Key	Description
datasource.ldap.pool.increment	When all connections in the pool are busy, the value of this property specifies the number of new connections to create and add to the pool.
datasource.ldap.pool.initialsize	The value of this property specifies the number of connections to be stored, at the time of creation and initialization of the pool.

Table 18-6 Connection Pools

UDDI Property Key	Description
datasource.ldap.pool.maxsize	The value of this property specifies the maximum number of connections that the pool may hold.
datasource.ldap.pool.systemmaxsize	The value of this property specifies the maximum number of connections created, even after the pool has reached its capacity. Once the pool reaches its maximum size, and all connections are busy, connections are temporarily created and returned to the client, but not stored in the pool. However once the system max size is reached, all requests for new connections are blocked until a previously busy connection becomes available.

Table 18-7 LDAP Datastore Configuration

UDDI Property Key	Description
datasource.ldap.manager.uid	The value of this property specifies back-end LDAP server administrator or privileged user ID, (e.g. cn=Directory Manager) who can save data in LDAP.
datasource.ldap.manager.password	The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for data storage.
datasource.ldap.server.url	The value of this property is an "ldap://" URL to the LDAP directory used for data storage.
datasource.ldap.server.root	The value of this property is the root entry of the LDAP directory used for data storage (e.g., dc=acumenat, dc=com).

Note: In a replicated LDAP environment, there are "m" LDAP masters and "n" LDAP replicas, respectively numbered from 0 to (m-1) and from 0 to (n-1). The fifth part of the property keys below, quoted as "i", refers to this number and differs for each LDAP server instance defined.

Table 18-8 Replicated LDAP Datastore Configuration

UDDI Property Key	Description
datasource.ldap.server.master.i.manager.uid	The value of this property specifies the administrator or privileged user ID for this "master" LDAP server node, (e.g. cn=Directory Manager) who can save data in LDAP.
datasource.ldap.server.master.i.manager.password	The value of this property is the password for the matching above user ID, and is used to establish connections with the relevant "master" LDAP directory to write data.
datasource.ldap.server.master.i.url	The value of this property is an "ldap://" URL to the corresponding LDAP directory node.
datasource.ldap.server.master.i.root	The value of this property is the root entry of the corresponding LDAP directory node (e.g., dc=acumenat, dc=com).
datasource.ldap.server.replica.i.manager.uid	The value of this property specifies the user ID for this "replica" LDAP server node, (e.g. cn=Directory Manager) who can read the UDDI data from LDAP.
datasource.ldap.server.replica.i.manager.password	The value of this property is the password for the matching above user ID, and is used to establish connections with the relevant "replica" LDAP directory to read data.
datasource.ldap.server.replica.i.url	The value of this property is an "ldap://" URL to the corresponding LDAP directory node.
datasource.ldap.server.replica.i.root	The value of this property is the root entry of the corresponding LDAP directory node (e.g., dc=acumenat, dc=com).

Table 18-9 File Datastore Configuration

UDDI Property Key	Description
datasource.file.directory	The value of this property specifies the directory where UDDI data is stored in the file system.

Table 18-10 General Security Configuration

UDDI Property Key	Description
security.custom.group.operators	The value of this property specifies a security group name, where the members of this group will be treated as UDDI administrators.

Table 18-11 LDAP Security Configuration

UDDI Property Key	Description
security.custom.ldap.manager.uid	The value of this property specifies security LDAP server administrator or privileged user ID, i.e. cn=Directory Manager who can save data in LDAP.
security.custom.ldap.manager.password	The value of this property is the password for the above user ID, and is used to establish connections with the LDAP directory used for security.
security.custom.ldap.url	The value of this property is an "ldap://" URL to the LDAP directory used for security.
security.custom.ldap.root	The value of this property is the root entry of the LDAP directory used for security (e.g., dc=acumenat, dc=com).

Table 18-11 LDAP Security Configuration

UDDI Property Key	Description
security.custom.ldap.userroot	The value of this property specifies the users root entry on the security LDAP server. For example, ou=People.
security.custom.ldap.group.root	The value of this property specifies the operator entry on the security LDAP server. For example, "cn=UDDI Administrators, ou=Groups". This entry contains IDs of all UDDI administrators.

Table 18-12 File Security Configuration

UDDI Property Key	Description
security.custom.file.userdir	The value of this property specifies the directory where UDDI security information (users and groups) is stored in the file system.

UDDI Directory Explorer

The UDDI Directory Explorer allows authorized users to publish Web Services in private WebLogic Server UDDI registries and to modify information for previously published Web Services.

The UDDI Directory Explorer also enables you to search both public and private UDDI registries for Web Services and information about the companies and departments that provide these Web Services. The Directory Explorer also provides access to details about the Web Services and associated WSDL files (if available.)

To invoke the UDDI Directory Explorer in your browser, enter the following URL:

```
http://host:port/uddiexplorer
```

where

- *host* refers to the computer on which WebLogic Server is running.
- *port* refers to the port number where WebLogic Server is listening for connection requests. The default port number is 7001.

You can perform the following tasks with the UDDI Directory Explorer:

- Search public registries
- Search private registries
- Publish to a private registry
- Modify private registry details
- Setup UDDI directory explorer

For more information about using the UDDI Directory Explorer, click the **Explorer Help** link on the main page.

UDDI Client API

WebLogic Server includes an implementation of the client-side UDDI API that you can use in your Java client applications to programmatically search for and publish Web Services.

The two main classes of the UDDI client API are `Inquiry` and `Publish`. Use the `Inquiry` class to search for Web Services in a known UDDI registry and the `Publish` class to add your Web Service to a known registry.

WebLogic Server provides an implementation of the following client UDDI API packages:

- `weblogic.uddi.client.service`
- `weblogic.uddi.client.structures.datatypes`
- `weblogic.uddi.client.structures.exception`
- `weblogic.uddi.client.structures.request`
- `weblogic.uddi.client.structures.response`

For detailed information on using these packages, see the UDDI API Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Pluggable tModel

A taxonomy is basically a tModel used as reference by a categoryBag or identifierBag. A major distinction is that in contrast to a simple tModel, references to a taxonomy are typically checked and validated. WebLogic Server's UDDI Server takes advantage of this concept and extends this capability by introducing custom taxonomies, called "pluggable tModels". Pluggable tModels allow users (UDDI administrators) to add their own checked taxonomies to the UDDI registry, or overwrite standard taxonomies.

To add a pluggable tModel:

1. Create an XML file conforming to the specified format described in "XML Schema for Pluggable tModels" on page 18-18, for each tModelKey/categorization.
2. Add the comma-delimited, fully qualified file names to the `pluggableTModel.file.list` property in the `uddi.properties` file used to configure UDDI Server. For example:

   ```
   pluggableTModel.file.list=c:/temp/cat1.xml,c:/temp/cat2.xml
   ```

   See "Configuring the UDDI 2.0 Server" on page 18-5 for details about the uddi.properties file.
3. Restart WebLogic Server.

The following sections include a table detailing the XML elements and their permissible values, the XML schema against which pluggable tModels are validated, and a sample XML.

XML Elements and Permissible Values

The following table describes the elements of the XML file that describes your pluggable tModels.

Table 18-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
Taxonomy	Required	Root Element		
checked	Required	Whether this categorization is checked or not.	true / false	If false, keyValue will not be validated.
type	Required	The type of the tModel.	categorization / identifier / valid values as defined in uddi-org-types	See uddi-org-types tModel for valid values.

Table 18-13 Description of the XML Elements to Configure Pluggable tModels

Element/Attribute	Required	Role	Values	Comments
applicability	Optional	Constraints on where the tModel may be used.		No constraint is assumed if this element is not provided
scope	Required if the applicability element is included.		businessEntity / businessService / bindingTemplate / tModel	tModel may be used in tModelInstanceInfo if scope "bindingTemplate" is specified.
tModel	Required	The actual tModel, according to the UDDI data structure.	Valid tModelKey must be provided.	
categories	Required if checked is set to true.			
category	Required if element categories is included	Holds actual keyName and keyValue pairs.	keyName / keyValue pairs	category may be nested for grouping or tree structure.
keyName	Required			
keyValue	Required			

XML Schema for Pluggable tModels

The XML Schema against which pluggable tModels are validated is as follows:

```
<simpleType name="type">
  <restriction base="string"/>
</simpleType>
```

```
<simpleType name="checked">
  <restriction base="NMTOKEN">
    <enumeration value="true"/>
    <enumeration value="false"/>
  </restriction>
</simpleType>

<element name="scope" type="string"/>

<element name = "applicability" type = "uddi:applicability"/>

<complexType name = "applicability">
  <sequence>
    <element ref = "uddi:scope" minOccurs = "1" maxOccurs = "4"/>
  </sequence>
</complexType>

<element name="category" type="uddi:category"/>

<complexType name = "category">
  <sequence>
   <element ref = "uddi:category" minOccurs = "0" maxOccurs = "unbounded"/>

  </sequence>
  <attribute name = "keyName" use = "required" type="string"/>
  <attribute name = "keyValue" use = "required" type="string"/>
</complexType>

<element name="categories" type="uddi:categories"/>

<complexType name = "categories">
  <sequence>
   <element ref = "uddi:category" minOccurs = "1" maxOccurs = "unbounded"/>
  </sequence>
</complexType>

<element name="Taxonomy" type="uddi:Taxonomy"/>

<complexType name="Taxonomy">
  <sequence>
    <element ref = "uddi:applicability" minOccurs = "0" maxOccurs = "1"/>
    <element ref = "uddi:tModel" minOccurs = "1" maxOccurs = "1"/>
    <element ref = "uddi:categories" minOccurs = "0" maxOccurs = "1"/>
  </sequence>
```

```
      <attribute name = "type" use = "required" type="uddi:type"/>
      <attribute name = "checked" use = "required" type="uddi:checked"/>
    </complexType>
```

Sample XML for a Pluggable tModel

The following shows a sample XML for a pluggable tModel:

```
<?xml version="1.0" encoding="UTF-8" ?>

  <SOAP-ENV:Envelope
      xmlns:SOAP_ENV="http://schemas.xmlsoap.org/soap/envelope/"
      xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
      xmlns:xsd="http://www.w3.org/1999/XMLSchema">

  <SOAP-ENV:Body>

   <Taxonomy checked="true" type="categorization" xmlns="urn:uddi-org:api_v2" >
    <applicability>
        <scope>businessEntity</scope>
        <scope>businessService</scope>
        <scope>bindingTemplate</scope>
    </applicability>
    <tModel  tModelKey="uuid:C0B9FE13-179F-41DF-8A5B-5004DB444tt2" >
        <name> sample pluggable tModel </name>
        <description>used for test purpose only </description>
            <overviewDoc>
                <overviewURL>http://www.abc.com </overviewURL>
            </overviewDoc>
    </tModel>
    <categories>
        <category keyName="name1 " keyValue="1">
            <category keyName="name11" keyValue="12">
                <category keyName="name111" keyValue="111">
                    <category keyName="name1111" keyValue="1111"/>
                    <category keyName="name1112" keyValue="1112"/>
                </category>
                <category keyName="name112" keyValue="112">
                    <category keyName="name1121" keyValue="1121"/>
                    <category keyName="name1122" keyValue="1122"/>
                </category>
            </category>
        </category>
        <category keyName="name2 " keyValue="2">
            <category keyName="name21" keyValue="22">
                <category keyName="name211" keyValue="211">
                    <category keyName="name2111" keyValue="2111"/>
                    <category keyName="name2112" keyValue="2112"/>
```

```
                </category>
                <category keyName="name212" keyValue="212">
                    <category keyName="name2121" keyValue="2121"/>
                    <category keyName="name2122" keyValue="2122"/>
                </category>
            </category>
        </category>
    </categories>
   </Taxonomy>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

CHAPTER 19

Interoperability

The following sections provide an overview of what it means for Web Services to be interoperable and tips on creating Web Services that interoperate with each other as much as possible:

- "Overview of Interoperability" on page 19-1
- "Avoid Using Vendor-Specific Extensions" on page 19-2
- "Stay Current With the Latest Interoperability Tests" on page 19-2
- "Understand the Data Models of Your Applications" on page 19-3
- "Understand the Interoperability of Various Data Types" on page 19-3
- "Results of SOAPBuilders Interoperability Lab Round 3 Tests" on page 19-5
- "Interoperating With .NET" on page 19-5

Overview of Interoperability

A fundamental characteristic of Web Services is that they are interoperable. This means that a client can invoke a Web Service regardless of the client's hardware or software. In particular, interoperability demands that the functionality of a Web Service application be the same across differing:

- Application platforms, such as BEA WebLogic Server, IBM Websphere, or Microsoft .NET.

- Programming languages, such as Java, C++, C#, or Visual Basic.
- Hardware, such as mainframes, PCs, or peripheral devices.
- Operating systems, such as different flavors of UNIX or Windows.
- Application data models.

For example, an interoperable Web Service running on WebLogic Server on a Sun Microsystems computer running Solaris can be invoked from a Microsoft .NET Web Service client written in Visual Basic.

To ensure the maximum interoperability, WebLogic Server supports the following specifications and versions when generating your Web Service:

- HTTP 1.1 for the transport protocol
- XML Schema to describe your data
- WSDL 1.1 to describe your Web Service
- SOAP 1.1 for the message format

The following sections provide some useful interoperability tips and information when writing Web Service applications.

Avoid Using Vendor-Specific Extensions

Avoid using vendor-specific implementation extensions to specifications (such as SOAP, WSDL, and HTTP) that are used by Web Services. If your Web Service relies on this extension, a client application that invokes it might not use the extension and the invoke might fail.

Stay Current With the Latest Interoperability Tests

Public interoperability tests provide information about how different vendor implementations of Web Service specifications interoperate with each other. This information is very useful if you are creating a Web Service on WebLogic Server that has to, for example, interoperate with Web Services from other vendors, such as .NET.

Warning: BEA's participation in these interoperability tests does not imply that BEA officially certifies its Web Services implementation against the other platforms participating in the tests.

The following Web sites include public interoperability tests:

- Web Service Interoperability Organization at http://www.ws-i.org/
- SoapBuilder Interoperability Lab at http://www.whitemesa.com/

You can also use the vendor implementations listed in these Web sites to exhaustively test your Web service for interoperability.

Understand the Data Models of Your Applications

A good use of Web Services is to provide a cross-platform technology for integrating existing applications. These applications typically have very different data models which your Web Service must reconcile.

For example, assume that you are creating a Web Service application to integrate the two accounting systems in a large company. Although the data models of each accounting system are probably similar, they most likely differ in at least some way, such as the name of a data field, the amount of information stored about each customer, and so on. It is up to the programmer of the Web Service to understand each data model, and then create an intermediate data model to reconcile the two. Typically this intermediate data model is expressed in XML using XML Schema. If you base your Web Service application on only one of the data models, the two applications probably will not interoperate very well.

Understand the Interoperability of Various Data Types

The data types of the parameters and return values of your Web Service operations have a great impact on the interoperability of your Web Service. The following table describes how interoperable the various types of data types are.

Table 19-1 Interoperability of Various Types of Data Types

Data Type	Description
JAX-RPC built-in data types	Interoperate with no additional programming. The JAX-RPC specification defines a subset of the XML Schema built-in data types that any implementation of JAX-RPC must support. Because all of these data types map directly to a SOAP-ENC data type, they are interoperable.
Built-in WebLogic Server data types	Interoperate with no additional programming. WebLogic Server includes support for all the XML Schema built-in data types. Because all of these data types map directly to a SOAP-ENC data type, they are interoperable. For the full list of built-in WebLogic Server data types, see "Supported Built-In Data Types" on page 5-15.
Non-built-in data types	Interoperate with additional programming or tools support. If your Web Service uses non-built-in data types, you must create the XML Schema that describes the XML representation of the data, the Java class that describes the Java representation, and the serialization class that converts the data between its XML and Java representation. WebLogic Server includes the `servicegen` and `autotype` Ant tasks that automatically generate these objects. Keep in mind, however, that these Ant tasks might generate an XML Schema that does not interoperate well with client applications or it might not be able to create an XML Schema at all if the Java data type is very complex. In these cases you might need to manually create the objects needed by non-built-in data types, as described in Chapter 11, "Using Non-Built-In Data Types." Additionally, you must ensure that client applications that invoke your Web Service include the serialization class needed to convert the data between its XML representation and the language-specific representation of the client application. WebLogic Server can generate the serialization class for Weblogic client applications with the `clientgen` Ant task. If, however, the client applications that invoke your Web Service are not written in Java, then you must create the serialization class manually.

Results of SOAPBuilders Interoperability Lab Round 3 Tests

For the results of WebLogic Web services' participation in the SOAPBuilders Interoperability Lab Round 3 tests, see http://webservice.bea.com:7001. The tests were run with version 8.1 of WebLogic Server.

For the test results, see http://webservice.bea.com/index.html#qz41; for the source code of the tests, see http://webservice.bea.com/index.html#qz40.

For more information on the SOAPBuilder Interoperability tests, see http://www.whitemesa.com.

Warning: BEA's participation in these interoperability tests does not imply that BEA officially certifies its Web Services implementation against the other platforms participating in the tests.

Interoperating With .NET

You invoke a .NET Web Service from a WebLogic Web Services client application exactly as described in Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server." When you execute the `clientgen` Ant task to generate the Web Service-specific client JAR file, use the `wsdl` attribute to specify the URL of the WSDL of the deployed .NET Web Service.

To invoke a deployed WebLogic Web Service from a .NET client application, use Microsoft Visual Studio .NET to create an application, then add a Web Reference, specifying the WSDL of the deployed WebLogic Web Service, as described in the following example. In Microsoft Visual Studio, adding a Web Reference is equivalent to executing the WebLogic `clientgen` Ant task.

Warning: The following example describes one way to invoke a WebLogic Web Service from a .NET client application. For the most current and detailed information about using Microsoft Visual Studio .NET to invoke WebLogic (and other) Web Services, consult the Microsoft documentation at http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vsintro7/html/vxconATourOfVisualStudio.asp.

1. Start and use Microsoft Visual Studio .NET to create your application as usual.

2. In the Solution Explorer in the right pane, right-click your application and chose Add Web Reference. The Solution Explorer Browser appears.
3. Enter the WSDL of the deployed WebLogic Web Service in the Solution Explorer Browser. As soon as the browser accepts the WSDL, the Add Reference button becomes active.

   See "WebLogic Web Services Home Page and WSDL URLs" on page 6-21 for information on getting the WSDL of a deployed WebLogic Web Service.
4. Click the Add Reference button. The WebLogic Web Service appears in the Solution Explorer.
5. In your application component that will be used to invoke the Web Service, such as a button, add Visual C# or Visual Basic code to invoke a particular operation of the Web Service. Visual Studio .NET uses statement completion to help you write this code. The following Visual C# code excerpt shows a simple example of invoking the `echoString` operation of the `SoapInteropBaseService` Web Service:

```
WebReference1.SoapInteropBaseService s = new SoapInteropBaseService();
string s = s.echoString("Hi there!");
```

   In the example, `WebReference1` is the name of the Web Reference you added in preceding steps.

CHAPTER 20

Troubleshooting

The following sections describe how to troubleshoot WebLogic Web Services:

- "Using the Web Service Home Page to Test Your Web Service" on page 20-2
- "Viewing SOAP Messages" on page 20-4
- "Posting the HTTP SOAP Message" on page 20-5
- "Debugging Problems with WSDL" on page 20-8
- "Verifying a WSDL File" on page 20-9
- "Verifying an XML Schema" on page 20-10
- "Debugging Data Type Generation (Autotyping) Problems" on page 20-10
- "Debugging Performance Problems" on page 20-11
- "Performance Hints" on page 20-12
- "Re-Resolving IP Addresses in the Event of a Failure" on page 20-12
- "BindingException When Running clientgen or autotype Ant Task" on page 20-13
- "Client Error When Using the WebLogic Web Service Client to Connect to a Third-Party SSL Server" on page 20-13
- "Client Error When Invoking Operation That Returns an Abstract Type" on page 20-14

- "Including Nillable, Optional, and Empty XML Elements in SOAP Messages" on page 20-15
- "SSLKeyException When Trying to Invoke a Web Service Using HTTPS" on page 20-17
- "Autotype Ant Task Not Generating Serialization Classes for All Specified Java Types" on page 20-17
- "Client Gets HTTP 401 Error When Invoking a Non-Secure Web Service" on page 20-18
- "Asynchronous Web Service Client Using JMS Transport Not Receiving Response Messages From WebLogic Server" on page 20-19
- "Running autotype Ant Task on a Large WSDL File Returns java.lang.OutOfMemoryError" on page 20-20
- "Error When Trying to Log Onto the UDDI Explorer" on page 20-20

Using the Web Service Home Page to Test Your Web Service

Every Web Service deployed on WebLogic Server has a Home Page. From the Home page you can:

- View the WSDL that describes the service.
- Test each operation with sample parameter values to ensure that it is working correctly.
- View the SOAP request and response messages from a successful execution of an operation.

URL Used to Invoke the Web Service Home Page

To invoke the Web Service Home page for a particular service in your browser, use the following URL:

```
[protocol]://[host]:[port]/[contextURI]/[serviceURI]
```

where:

- *protocol* refers to the protocol over which the service is invoked, either `http` or `https`. This value corresponds to the `protocol` attribute of the `<web-service>` element that describes the Web Service in the `web-services.xml` file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `protocol` attribute.
- *host* refers to the computer on which WebLogic Server is running.

- *port* refers to the port number on which WebLogic Server is listening (default value is `7001`).
- *contextURI* refers to the context root of the Web application, corresponding to the `<context-root>` element in the `application.xml` deployment descriptor of the EAR file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `contextURI` attribute.

  If your `application.xml` file does not include the `<context-root>` element, then the value of `contextURI` is the name of the Web application archive file or exploded directory.
- *serviceURI* refers to the URI of the Web Service. This value corresponds to the `uri` attribute of the `<web-service>` element in the `web-services.xml` file. If you used the `servicegen` Ant task to assemble your Web Service, this value corresponds to the `serviceURI` attribute.

For example, assume you used the following `build.xml` file to assemble a WebLogic Web Service using the `servicegen` Ant task:

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
      <servicegen
         destEar="myWebService.ear"
         warName="myWAR.war"
         contextURI="web_services">
         <service
             ejbJar="myEJB.jar"
             targetNamespace="http://www.bea.com/examples/Trader"
             serviceName="TraderService"
             serviceURI="/TraderService"
             generateTypes="True"
             expandMethods="True" >
         </service>
     </servicegen>
  </target>
</project>
```

The URL to invoke the Web Service Home Page, assuming the service is running on a host called `ariel` at the default port number, is:

```
http://ariel:7001/web_services/TraderService
```

Testing the Web Service

The Web Service Home Page lists the operations that can be invoked for this service. To test a particular operation:

1. Click on the operation link.
2. Enter sample values for the parameters in the table. The first two columns of the table list the name and Java data type of the operation.
3. Click Invoke.

   The SOAP request and response messages and the value returned by the operation are displayed in a new browser window.

The main Web Service Home Page also displays an example of the Java code to invoke one of the operations and a sample `build.xml` file for executing the `clientgen` Ant task to generate the Web Service-specific client JAR file.

Viewing SOAP Messages

If you encounter an error while trying to invoke a Web Service (either WebLogic or non-WebLogic), it is useful to view the SOAP request and response messages, because they often point to the problem.

To view the SOAP request and response messages, run your client application with the `-Dweblogic.webservice.verbose=true` flag, as shown in the following example that runs a client application called `my.app.RunService`:

```
prompt> java -Dweblogic.webservice.verbose=true my.app.RunService
```

The full SOAP request and response messages are printed in the command window from which you ran your client application.

You configure this feature by setting verbose mode to true, either with Ant or programmatically.

Setting Verbose Mode with Ant

If you use Ant to run your client application, you can set verbose mode by adding a `<sysproperty>` element to the `build.xml` file, as shown in the following example:

```
<java classname="my.app.RunService">
  <sysproperty key="weblogic.webservice.verbose" value="true"/>
</java>
```

You can also configure WebLogic Server to print the SOAP request and response messages each time a deployed WebLogic Web Service is invoked by specifying the `-Dweblogic.webservice.verbose=true` flag when you start WebLogic Server. The SOAP messages are printed to the command window from which you started WebLogic Server.

Note: Because of possible decrease in performance due to the extra output, BEA recommends you set this WebLogic Server flag only during the development phase.

Setting Verbose Mode Programatically

You can programmatically set verbose mode in your client application by using the `weblogic.webservice.binding.BindingInfo.setVerbose(true)` method, as shown in the following code excerpt:

```
import weblogic.webservice.binding.BindingInfo;

...

   BindingInfo info =
    (BindingInfo)stub._getProperty("weblogic.webservice.bindinginfo" );

   info.setVerbose( true );
   port.helloWorld();
```

In the example, `stub` is the instance of the JAX-RPC `Stub` class for your Web Service. When the `helloWorld()` operation executes, the SOAP request and response messages will be printed in the command window from which you executed the client application.

To turn off verbose mode, invoke the `setVerbose(false)` method.

For more information about the `weblogic.webservice.binding` package, see the Javadocs at http://e-docs.bea.com/wls/docs81/javadocs/index.html.

Note: The `weblogic.webservice.binding` package is a proprietary WebLogic API.

Posting the HTTP SOAP Message

To further troubleshoot problems with the SOAP messages, you can post the request directly to a SOAP server (rather than through a client application) and view the raw SOAP response. By-passing the client application and viewing the raw SOAP messages may pinpoint the problem. You can then update selected parts of the SOAP request by editing the text file, then re-post the request to see what fixes the problem.

Note: It is assumed that you understand the structure of a SOAP message; if you need more detailed information about the SOAP XML Schema, see SOAP 1.1 at http://www.w3.org/TR/SOAP.

To post a SOAP request to a SOAP server directly:

1. Create a text file that contains an HTTP SOAP request; the request should include both the HTTP headers and SOAP envelope. See "Composing the SOAP Request" on page 20-7 for more information on creating this file. The following example shows an HTTP SOAP request:

```
POST /asmx/simple.asmx
HTTP/1.1 Host: www.mssoapinterop.org
Content-Type: text/xml; charset=utf-8
Connection: close
SOAPAction: "http://soapinterop.org/"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:tns="http://soapinterop.org/"
   xmlns:types="http://soapinterop.org/encodedTypes"
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body
      soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
      <tns:echoString>
        <inputString xsi:type="xsd:string">string</inputString>
      </tns:echoString>
   </soap:Body>
</soap:Envelope>
```

2. Use the `weblogic.webservice.tools.debug.Post` utility to post the message to a SOAP server, as shown in the following example:

   ```
   java weblogic.webservice.tools.debug.Post filename
   ```

   where *`filename`* refers to the text file that contains the HTTP SOAP request, created in the preceding step. The Post utility uses the HTTP header to determine the URL of the SOAP server.

3. The SOAP server sends back the raw HTTP SOAP response which you can examine for clues about your problem.

Composing the SOAP Request

This section describes how to create a file that contains a well-formed HTTP SOAP request generated by the WebLogic Web Services client when invoking a Web Service.

1. Copy into a file the generated SOAP request for the invocation of a Web Service by either using the `weblogic.webservice.verbose` property, as described in "Viewing SOAP Messages" on page 20-4, or cutting and pasting the SOAP message generated from testing the WebLogic Web Service from its Home Page, as described in "Using the Web Service Home Page to Test Your Web Service" on page 20-2.

   The following SOAP request was generated f rom an invocation of the sample `examples.webservices.complex.statelessSession` Web Service , and was cut and pasted from the Web Services Home Page:

   ```
   <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <env:Header>
    </env:Header>
    <env:Body
   env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
     <m:sell    xmlns:m="http://www.bea.com/examples/Trader">
      <string     xsi:type="xsd:string">sample string</string>
      <intVal     xsi:type="xsd:int">100</intVal>
     </m:sell>
    </env:Body>
   </env:Envelope>
   ```

2. By default, the generated SOAP request does not include the standard XML declaration, so add the following line, shown in bold, to the beginning of the file:

   ```
   <?xml version="1.0" encoding="utf-8"?>
   <env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   ...
   ```

3. Ensure that the XML is well-formed by opening it in an XML editor, such as XMLSpy, and editing where necessary. XMLSpy is a product that is installed with BEA WebLogic Platform.

4. Add the needed HTTP headers to the beginning of the file, with the appropriate Host and POST header values, as shown in bold in the following example:

   ```
   POST /filetransferAtResponse/FTService HTTP/1.1
   Host: localhost:7001
   ```

```
Content-Type: text/xml; charset=utf-8
Connection: close
SOAPAction: ""

<?xml version="1.0" encoding="utf-8"?>
<env:Envelope  xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
...
```

HTTP is very strict about structure when parsing a request, so be sure you create a well-formed HTTP request. In particular, be sure you:

- Include a blank line between the headers and the XML declaration.
- Do *not* include any extra spaces after the headers.

You should now have a good HTTP SOAP request to post to a SOAP server.

Debugging Problems with WSDL

Another potential reason for errors produced when invoking a Web Service is that the WSDL might be invalid. Fixing a published WSDL that contains problems might be out of your hands; however, you can at least pinpoint the problem and let the provider know so that the provider can fix it.

Note: It is assumed that you understand the structure of a WSDL file; if you need more information on the WSDL XML Schema, see Web Services Description Language (WSDL) 1.1 at http://www.w3.org/TR/wsdl.

This section does not attempt to cover all possible problems with a WSDL file but rather, describe the following common ones:

- An invalid URL for the Web Service endpoint. This URL is listed in the `<service>` element of the WSDL, as shown in the following excerpt:

  ```
  <service name="myservice>
    <port name="myport" binding="tns:mybinding">
      <soap address="http://a_host:4321/service" />
    </port>
  </service>
  ```

  In this case, ensure that the URL `http://a_host:4321/service` is indeed the Web Service endpoint.

- References to undefined elements

  For example, the `type` attribute of the `<binding>` element refers to the `name` attribute of a `<portType>` element, as shown in the following excerpts:

```
<binding name="my-binding" type="tns:my-port">
  <operation name="foo">
     <input />
     <output />
  </operation>
</binding>

<portType name="my-port">
  <operation name="foo">
    <input message="tns:fooReq" />
    <output message="tns:fooRes" />
  </operation>
</portType>
```

If, for example, the `<portType>` element had a `name` of `my-port1`, then the reference to it in the `<binding>` element would be invalid, and attempting to invoke the Web Service described by this WSDL would fail.

- Problems with the `<import>` element.

  WSDL allows associating a namespace with a document location using an `<import>` element, as shown:

  ```
  <definitions .... >
     <import
       namespace="http://example.com/stockquote/definitions"
       location="http://example.com/stockquote/stockquote.wsdl"/>
   </definitions>
  ```

  The WSDL specified in the `location` attribute might itself have an import statement that points to another WSDL, and so on. Although this is a good technique for creating clearer Web Service definitions, because it separates the definitions according to their level of abstraction, it is also possible to create a problem if there are many layers of abstraction. In this case, make sure all imported WSDL files actually exist and are valid.

- The WSDL may contain XML data types that are not compatible with WebLogic Web Services.

Verifying a WSDL File

To verify that a WSDL is compatible with WebLogic Web Services, use the `clientgen` Ant task with the `wsdl` attribute, as shown in the following example:

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="myapps/myService_client.jar"
/>
```

If the `clientgen` Ant task completes with no errors, then the WSDL is compatible and well-formed.

Verifying an XML Schema

To verify that an XML Schema is compatible with WebLogic Web Services, use the `autotype` Ant task with the `schemaFile` attribute, as shown in the following example:

```
<autotype schemaFile="my-schema.xsd"
          packageName="foo"
          destDir="temp_dir"
/>
```

If the `autotype` Ant task completes with no errors, then the XML Schema is compatible and well-formed.

Debugging Data Type Generation (Autotyping) Problems

If you encounter an error while using the `servicegen`, `autotype`, or `clientgen` Ant tasks to generate the autotyping components (such as the serialization class and Java or XML representations) for any non-built-in data types, you can set the `weblogic.xml.schema.binding.verbose=true` property to print out verbose information about the autotyping activity taking place, and perhaps get an idea of what the problem is.

You can set this property while using the command-line versions of the `autotype` or `clientgen` Ant tasks, as shown in the following example:

```
java -Dweblogic.xml.schema.binding.verbose=true \
   weblogic.webservice.clientgen -wsdl foo.wsdl   \
   -clientJar /tmp/test_client.jar -packageName foo
```

Common XML Schema Problems

The following list describes typical problems with your XML Schema when using the autotyping features of WebLogic Server (in other words, the `autotype`, `servicegen`, or `clientgen` Ant tasks) to generate the serialization class and Java representation of a non-built-in XML data type:

- A missing import statement for the namespace associated with a data type.
- Using unsupported XML Schema data types. See "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16 and "Unsupported Features" on page 1-11 for more information.

Common Java Problems

The following list describes typical problems with your Java class when using the autotyping features of WebLogic Server (in other words, the `autotype`, `servicegen`, or `clientgen` Ant tasks) to generate the serialization class and XML Schema representation of a non-built-in Java data type:

- The Java class does not have a public default constructor.
- The Java class does not have *both* get and set methods for all private fields. In this case, the autotyping feature of the Web Services Ant tasks will ignore these private fields when generating the serialization class and corresponding XML Schema.

  If you use public fields in your Java class, you do not have to create get and set methods for each field.
- Using unsupported Java data types. For the full list of the non-built-in Java data types that the autotyping feature supports, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16
- Not being able to roundtrip generated Java/XML data types. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

Debugging Performance Problems

Web Services use SOAP as their message protocol. Other binary protocols will likely achieve better performance. For example, if you can invoke a Web Service 300 times a second, you might be able to invoke the same method 1500 times a second using RMI.

The main factors that determine the performance of a Web Service, from the most influential to the least, are as follows:

- Using HTTP as the connection protocol
- If you are using security, the process of encrypting and decrypting the SOAP message
- Parsing and generating XML, such as the SOAP message
- Converting data between its Java and XML representations
- Using very large parameters

Typically, HTTP has the most influence in the performance of a Web Service. To determine if this is true for your WebLogic Web Service, follow these guidelines:

1. Create a servlet which simply receives the SOAP message that is used to invoke your Web Service and returns the SOAP response message. Your servlet should do no other processing, such as converting data between XML and Java. For details on getting the SOAP request and response, see "Viewing SOAP Messages" on page 20-4.
2. Time how long it takes to invoke the Web Service in the standard way.
3. Time how long it takes to send the SOAP request to the servlet and for your client to receive the response.
4. Invoking the Web Service in the standard way should take only a little longer than sending the SOAP messages to the servlet. If this is true for your Web Service, then there is not much more you can do to speed up the invoke because HTTP is the main factor. However, if it takes a lot more time (such as twice as long) to invoke the Web Service than it does to use the servlet, then you might be running into one of the other factors. See "Performance Hints" on page 20-12 for information on how to increase the performance of your Web Service.

Performance Hints

The following list describes performance issues you should be aware of as you program your WebLogic Web Service.

- Use the `us-ascii` character set whenever you can, because it is the most efficient and fast. For details, see "Specifying the Character Set for a WebLogic Web Service" on page 14-2.
- Use literal encoding rather than SOAP encoding by specifying that your Web Service be document-oriented. For details, see "Choosing RPC-Oriented or Document-Oriented Web Services" on page 4-3.
- Security, such as data encryption and digital signatures, can slow down performance significantly, so be very judicious when adding security to a Web Service.
- Be very aware of what the handlers in your handler chains are doing, because they will execute for every single Web Service operation invoke.
- Be sure to turn off all debugging flags you might have turned on during development.

Re-Resolving IP Addresses in the Event of a Failure

The first time you invoke a Web Service from a client application that uses the WebLogic client JAR files, the client caches the IP address of the computer on which the Web Service is running, and by default this cache is never refreshed with a new DNS lookup. This means that if you

invoke a Web Service, and later the computer on which the Web Service is running crashes, but then another computer with a different IP address takes over for the crashed computer, a subsequent invoke of the Web Service from the original client application will fail because the client application continues to think that the Web Service is running on the computer with the old cached IP address. In other words, it does not try to re-resolve the IP address with a new DNS lookup, but rather uses the cached information from the original lookup.

To work around this problem, update your client application to set the JDK 1.4 system property `sun.net.inetaddr.ttl` to the number of seconds that you want the application to cache the IP address.

BindingException When Running clientgen or autotype Ant Task

If you use the `clientgen` or `autotype` Ant tasks with the `wsdl` attribute to generate client or data type components from a WSDL file, you might sometimes get the following exception:

```
weblogic.webservice.tools.build.WSBuildException: Failed to do type mapping -
with nested exception:
[weblogic.xml.schema.binding.BindingException: unable to find a definition for
type datatype
```

This exception means that there is an undefined data type in the section of the WSDL file that describes the XML Schema data types used by the Web Service. The solution to this problem is to add the data type definition to the WSDL file.

Client Error When Using the WebLogic Web Service Client to Connect to a Third-Party SSL Server

You can use the WebLogic client-side implementation of SSL in your client application to connect to a third-party SSL server, such as OpenSSL, by specifying the `weblogic.webservice.client` https protocol handler, as shown in the following example:

```
-Djava.protocol.handler.pkgs=weblogic.webservice.client
```

However, because of the way that the WebLogic client-side SSL was implemented, you must use the `SSLAdapter` class to open a URL connection to the SSL server and get an `InputStream`, as shown in the following code snippet:

```
SSLAdapter adapter =
    SSLAdapterFactory.getDefaultFactory().getSSLAdapter();
InputStream in = adapter.openConnection(url).getInputStream();
```

The preceding code replaces generic code to open a connection, shown in the following example:

```
URLConnection con = url.openConnection();
InputStream in = con.getInputStream();
```

If you do not use the `SSLAdapter` class as shown, you might get the following error when running your client:

```
Exception: FATAL Alert:BAD_CERTIFICATE - A corrupt or unuseable
certificate was received
```

Client Error When Invoking Operation That Returns an Abstract Type

When a client invokes a Web Service operation implemented with a method that returns an abstract type, the client might get the following error:

```
java.lang.Error: cannot create abstract type: my.abstractType
```

The exact scenario for this error to occur is as follows:

```
abstract class Foo { }

class Bar extends Foo {}

class MyService {
  public Foo getFoo() {
    return new Bar();
  }
}
```

So, although the signature of the `getFoo()` method specifies that it returns a `Foo` object, the actual `return` statement in the implemenation of the method returns a `Bar` object, which extends the abstract `Foo`.

In this scenario, it is important that you explicitly execute the `autotype` Ant task for the `Bar` class to generate its serialization components *before* you execute `autotype` on the `MyService` class. The second `autotype` execution on the `MyService` class automatically generates serialization components for the `Foo` abstract class because it is the explicit return value of the `getFoo()` method. If you execute the two `autotype` tasks in the reverse order, you will get an error when trying to invoke the Web Service operation that is implemented by the `getFoo()` method, even though you will not get an error when executing the Ant tasks themselves.

The following snippet from a `build.xml` file shows an example of running the two `autotype` Ant tasks in the correct order:

```
<autotype

      javaTypes="Bar"
      targetNamespace="com.bea.example"
```

```
    packageName="com.bea.example"
    keepGenerated="True"
    destDir="${classes}">
    <classpath>
      <path refid="project.classpath"/>
      <pathelement path="${classes}"/>
    </classpath>
  </autotype>

  <autotype
    javaComponents="MyService"
    targetNamespace="com.bea.example"
    typeMappingFile="${classes}/types.xml"
    packageName="com.bea.example"
    keepGenerated="True"
    destDir="${classes}">
    <classpath>
      <path refid="project.classpath"/>
      <pathelement path="${classes}"/>
    </classpath>
  </autotype>
```

Note: You cannot use the `servicegen` Ant task on the `MyService` class to generate all the serialization components in this scenario. This is because the `servicegen` Ant task will not know to generate components for the `Bar` class, because this class does not explicitly appear in the signatures of the methods of the `MyService` class.

Including Nillable, Optional, and Empty XML Elements in SOAP Messages

When WebLogic Server generates the SOAP response to an invocation of a Web Service operation, and one of the XML elements of the return value is defined as nillable and optional (or in other words, the XML Schema definition of the element includes the `nillable="true"` and `minOccurs="0"` attributes), and there is no actual data associated with the element, then WebLogic Server does not include the element in the SOAP response at all. This behavior, although not a bug in WebLogic Server, might be unexpected and could cause interoperability problems when different clients invoke the Web Service.

For example, assume the WSDL of your Web Service defines the `ProductType` XML data type as shown:

```
<xsd:complexType name="ProductType">

   <xsd:sequence>

    <xsd:element type="xsd:string" name="ID"
                 minOccurs="0" nillable="true"/>
```

```
      <xsd:element type="xsd:string" name="Name"
                   minOccurs="0" nillable="true"/>
      <xsd:element type="xsd:string" name="Description"
                  minOccurs="0" nillable="true"/>
    </xsd:sequence>
  </xsd:complexType>
```

Further assume a Web Service operation returns a `Product`, which is of type `ProductType`, and that in a particular invocation, the `Description` element is empty because the product has no description. WebLogic Server generates a SOAP response similar to the following:

```
<env:Envelope xmlns:env="http://schemas.xmlsoap.org/soap/envelope/";
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance";
          xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/";
          xmlns:xsd="http://www.w3.org/2001/XMLSchema";>
  <env:Header/>
    <env:Body>
      <n1:Product xmlns:n1="http://mycompany.com/mywebservice"; >
         <n1:ID>1234</n1:ID>
         <n1:Name>MyFabProduct</n1:Name>
      </n1:Product>
    </env:Body>
</env:Envelope>
```

Note that the `<n1:Product>` element simply does not include the `<n1:Description>` child element at all.

This behavior is different if the XML element is not optional (`minOccurs="1"`). In this case, WebLogic Server includes the empty element in the SOAP response, but with the `xsi:nil="true"` attribute, as shown in the following example:

```
...
      <n1:Product xmlns:n1="http://mycompany.com/mywebservice"; >
         <n1:ID>1234</n1:ID>
         <n1:Name>MyFabProduct</n1:Name>
         <n1:Description xsi:nil="true"></n1:Description>
      </n1:Product>
...
```

This is not a bug in WebLogic Server. The difference in behavior is due to the ambiguity of the XML Schema Part 0: Primer specification, which is very clear about what should happen when `minOccurs="1"`, but unclear in the case where `minOccurs="0"`.

If you always want nillable and optional XML elements to appear in the SOAP response, even when they have no content, then you can do one of the following:

- Explicitly set the value of the corresponding Java object to `null`, using a `setXXX(null)` method, in the backend implementation of your Web Service operation.

- Update the WSDL of the Web Service so that all `nillable="true"` XML elements that might sometimes be empty also have the `minOccurs="1"` attribute set. This option is not always possible, however, so BEA recommends the preceding workaround.

SSLKeyException When Trying to Invoke a Web Service Using HTTPS

When a client application invokes, for the first time, a Web Service whose endpoint URL uses HTTPS, the application might get the following error:

```
[java]  </bea_fault:stacktrace>javax.net.ssl.SSLKeyException: FATAL
Alert:BAD_CERTIFICATE - A corrupt or unuseable certificate was received.
```

This can happen when, for example, you initially ran the `clientgen` Ant task to generate the stubs from a WSDL whose endpoint address uses HTTP, create a client application that invokes this Web Service, and then switch to an endpoint address that uses HTTPS (and thus SSL) in the client application by setting the `ENDPOINT_ADDRESS_PROPERTY` property of the `javax.xml.rpc.Stub` interface, as shown in the following example:

```
String url = "https://localhost:7002/webservice/TraderService";
((javax.xml.rpc.Stub )trader)._setProperty
(javax.xml.rpc.Stub.ENDPOINT_ADDRESS_PROPERTY, url);
```

The problem in this case could be that the client application is not using the `WLSSLAdapter` class to load the client certificate, which is needed for SSL. The problem only arises when using HTTPS, which is why the problem did not occur when invoking the Web Service using HTTP.

To solve the problem, use the `WLSSLAdapter.setTrustedCertificateFile()` method (for 1-way SSL) or `WLSSLAdapter.loadLocalIdentity()` method (for 2-way SSL) to load the client certificate, as shown in the following example:

```
SSLAdapterFactory factory = SSLAdapterFactory.getDefaultFactory();
WLSSLAdapter adapter = (WLSSLAdapter) factory.getSSLAdapter();

// Uncomment following to load the client certificate for 1-way SSL
// adapter.setTrustedCertifcatesFile("mytrustedcerts.pem");

// Uncomment following to load the client certificate for 2-way SSL
// adapter.loadLocalIdentity(clientCredentialFile, pwd.toCharArray());
```

Autotype Ant Task Not Generating Serialization Classes for All Specified Java Types

When you use the `autotype` Ant task to generate serialization classes for a list of Java data types whose class names are the same, but are in different packages, make sure you do *not* specify the

packageName attribute. If you do, the autotype Ant task generates the serialization class for only the last Java data type, rather than all the specified Java data types.

For example, assume you want to generate serialization classes for the following Java data types:

- mypackage.MyClass
- mypackage.test.MyClass

The following sample autotype Ant task specification in the build.xml file is correct and will generate serialization classes for the two Java data types:

```
<autotype
    destDir="/output/type_defs"
    javaTypes="mypackage.MyClass,mypackage.test.MyClass"
    keepGenerated="True"
    overwrite="True">
    <classpath refid="client.classpath"/>
</autotype>
```

The following autotype specification is incorrect and will generate only one serialization class (for the mypackage.test.MyClass class):

```
<autotype
    destDir="/output/type_defs"
    javaTypes="mypackage.MyClass,mypackage.test.MyClass"
    keepGenerated="True"
    overwrite="True"
    packageName="mypackage">
    <classpath refid="client.classpath"/>
</autotype>
```

Client Gets HTTP 401 Error When Invoking a Non-Secure Web Service

If a client application includes the Authorization HTTP header in its SOAP request when invoking a Web Service, but the Web Service has *not* been configured with access control security constraints, WebLogic Server still refuses the request with an HTTP 401 Error: Unauthorized Access. This differs from the way Web Applications handle the same situation: Web Applications ignore the Authorization HTTP header if the Web Application is not configured with security constraints.

If you want your Web Service to behave like a Web Application in this situation, set the ignoreAuthHeader="True" attribute of the servicegen or source2wsdd Ant task that assembles your Web Service, as shown in the following example:

```
<servicegen
    destEar="ears/myWebService.ear"
    warName="myWAR.war">
    <service
      javaClassComponents="examples.webservices.basic.javaclass.HelloWorld"
      targetNamespace="http://www.bea.com/examples/HelloWorld"
      serviceName="HelloWorld"
      serviceURI="/HelloWorld"
      generateTypes="True"
      ignoreAuthHeader="True"
      expandMethods="True">
    </service>
</servicegen>
```

Setting this attribute in the Ant task in turn sets the `ignoreAuthHeader="True"` attribute for the `<web-service>` element that describes the Web Service in the generated `web-services.xml` deployment descriptor.

Warning: Be careful using the `ignoreAuthHeader` attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it.

Asynchronous Web Service Client Using JMS Transport Not Receiving Response Messages From WebLogic Server

You can configure a WebLogic Web Service so that client applications can use the JMS transport to invoke the Web Service. This feature is described in Chapter 9, "Using JMS Transport to Invoke a WebLogic Web Service." Furthermore, you can write a client application to invoke an operation of a Web Service asynchronously, which means that the client application first invokes the operation without immediately waiting for the result, and then optionally gets the results of the invoke in a later step. This feature is described in "Writing an Asynchronous Client Application" on page 7-11.

However, be aware that if you use the two features together, in certain situations the client application might never receive the asynchronous response message from WebLogic Server that includes the results of an initial invoke of the Web Service operation. In particular, assume that an asynchronous client application invokes an operation, but before the application can invoke the second request for the results of the operation, WebLogic Server is restarted. After WebLogic Server starts up again, it sees that it has a response message to send back to the client, but it does

not know where to send this response, and thus the asynchronous client application never receives it. This is because the Web Service asynchronous client uses temporary, rather than permanent, JMS destinations in its implementation, and references to this temporary destination from WebLogic Server are lost after a server restart.

Running autotype Ant Task on a Large WSDL File Returns java.lang.OutOfMemoryError

If you run the `autotype` Ant task on a very large WSDL file, your computer might run out of resources and return any one of the following errors:

```
The system is out of resources.
Consult the following stack trace for details. java.lang.OutOfMemoryError
package weblogic.xml.schema.binding.internal.builtin does not exist
```

To solve this problem, expand the memory of the `java` command used by the Ant task by increasing the heap size to at least 512M.

In particular, update the `ant.bat` file, located in the *`BEA_HOME`*`/weblogic81/server/bin` directory, where *`BEA_HOME`* is the main BEA installation directory, such as `c:/bea`. Update the file by adding the `-Xmx512m` option to the `%_JAVACMD%` variable used in the various `:runAnt` labels . For example:

```
:runAnt

"%_JAVACMD%" -Xmx512m -classpath "%LOCALCLASSPATH%" -Dant.home="%ANT_HOME%"
 %ANT_OPTS% org.apache.tools.ant.Main %ANT_ARGS% %ANT_CMD_LINE_ARGS%

if errorlevel 1 exit /b 1

goto end
```

Error When Trying to Log Onto the UDDI Explorer

If your WebLogic Server domain was created by a user different from the user that installed WebLogic Server, the following error is returned when a user tries to log onto the UDDI Explorer:

```
An error has occurred

E_fatalError(10500): a serious technical error has occurred while
processing the request. 'Exception while attempting to instantiate
subclass of DataReader: com.acumenat.uddi.persistence.ldap.LDAPInit'
```

To resolve this problem, the WebLogic Server administrator must change the permissions on the `uddi.properties` file to give access to all users. The `uddi.properties` file, used to configure the UDDI server, is located in the *`WL_HOME`*`/server/lib` directory, where *`WL_HOME`* refers to the main WebLogic Platform installation directory.

CHAPTER 21

Upgrading WebLogic Web Services

The following sections describe how to upgrade WebLogic Web Services to 8.1:

- "Overview of Upgrading WebLogic Web Services" on page 21-1
- "Upgrading a 7.0 WebLogic Web Service to 8.1" on page 21-1
- "Upgrading a 6.1 WebLogic Web Service to 8.1" on page 21-2

Overview of Upgrading WebLogic Web Services

Because of changes in the Web Service runtime system between Versions 6.1, 7.0, and 8.1 of WebLogic Server, you must upgrade Web Services created in version 6.1 and 7.0 to run on Version 8.1.

Upgrading a 7.0 WebLogic Web Service to 8.1

To upgrade a 7.0 WebLogic Web Service to Version 8.1:

1. Set your 8.1 environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is

BEA_HOME/user_projects/domains/*domainName*, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Change to the staging directory that contains the components of your Version 7.0 Web Service, such as the EJB JAR file and the `build.xml` file that contains the call to the `servicegen` Ant task.

3. Execute the `servicegen` Ant task specified in the `build.xml` file by typing `ant` in the staging directory:

   ```
   prompt> ant
   ```

   The Ant task generates the 8.1 Web Services EAR file in the staging directory which can then deploy on WebLogic Server.

Upgrading a 6.1 WebLogic Web Service to 8.1

You upgrade a 6.1 Web Service manually, by rewriting the `build.xml` file you used to create the 6.1 Web Service to now call the `servicegen` Ant task rather than the `wsgen` Ant task. You cannot deploy a 6.1 Web Service on a 8.1 WebLogic Server instance.

Warning: The `wsgen` Ant task was deprecated in Version 7.0 of WebLogic Server, and is not supported in Version 8.1.

The WebLogic Web Services client API included in version 6.1 of WebLogic Server has been removed and you cannot use it to invoke 8.1 Web Services. Version 8.1 includes a new client API, based on the Java API for XML based RPC (JAX-RPC). You must rewrite client applications that used the 6.1 Web Services client API to now use the JAX-RPC APIs. For details, see Chapter 7, "Invoking Web Services from Client Applications and WebLogic Server."

To upgrade a 6.1 WebLogic Web Service to 8.1:

1. Convert the `build.xml` Ant build file used to assemble 6.1 Web Services with the `wsgen` Ant task to the 8.1 version that calls the `servicegen` Ant task.

   For details see "Converting a 6.1 build.xml file to 8.1" on page 21-3.

2. Un-jar the 6.1 Web Services EAR file and extract the EJB JAR file that contains the stateless session EJBs (for 6.1 RPC-style Web Services) or message-driven beans (for 6.1 message-style Web Services), along with any supporting class files.

3. If your 6.1 Web Service was RPC-style, see"Assembling WebLogic Web Services Using the servicegen Ant Task" on page 6-3 for instructions on using the `servicegen` Ant task. If your 6.1 Web Service was message-style, see "Assembling JMS-Implemented WebLogic Web Services Using servicegen" on page 16-5.

4. In your client application, update the URL you use to access the Web Service or the WSDL of the Web Service from that used in 6.1 to 8.1. For details, see "Updating the URL Used to Access the Web Service" on page 21-5.

Converting a 6.1 build.xml file to 8.1

The main difference between the 6.1 and 8.1 `build.xml` files used to assemble a Web Service is the Ant task: in 6.1 the task was called `wsgen` and in 8.1 it is called `servicegen`. The `servicegen` Ant task uses many of the same elements and attributes of `wsgen`, although some do not apply anymore. The `servicegen` Ant task also includes additional configuration options. The table at the end of this section describes the mapping between the elements and attributes of the two Ant tasks.

The following `build.xml` excerpt is from the 6.1 RPC-style Web Services example:

```
<project name="myProject" default="wsgen">
        <target name="wsgen">
           <wsgen destpath="weather.ear"
                  context="/weather">
                <rpcservices path="weather.jar">
                   <rpcservice bean="statelessSession"
                               uri="/weatheruri"/>
                   </rpcservices>
           </wsgen>
        </target>
</project>
```

The following example shows an equivalent 8.1 `build.xml` file:

```
<project name="myProject" default="servicegen">
    <target name="servicegen">
       <servicegen
           destEar="weather.ear"
           contextURI="weather" >
           <service
               ejbJar="weather.jar"
               serviceURI="/weatheruri"
               includeEJBs="statelessSession" >
           </service>
       </servicegen>
```

```
    </target>
</project>
```

For detailed information on the WebLogic Web Service Ant tasks, see Appendix B, "Web Service Ant Tasks and Command-Line Utilities."

The following table maps the 6.1 `wsgen` elements and attributes to their equivalent 8.1 `servicegen` elements and attributes.

Table 21-1 6.1 to 8.1 wsgen Ant Task Mapping

6.1 wsgen Element	Attribute	Equivalent 8.1 servicegen element	Attribute
wsgen	basepath	No equivalent.	No equivalent
	destpath	servicegen	destEar
	context	servicegen	contextURI
	protocol	servicegen.service	protocol
	host	No equivalent.	No equivalent
	port	No equivalent.	No equivalent
	webapp	servicegen	warName
	classpath	servicegen	classpath
rpcservices	module	No equivalent.	No equivalent
	path	servicegen.service	ejbJar
rpcservice	bean	servicegen.service	includeEJBS, excludeEJBs
	uri	servicegen.service	serviceURI
messageservices	N/A	No equivalent.	No equivalent

Table 21-1 6.1 to 8.1 wsgen Ant Task Mapping

6.1 wsgen Element	Attribute	Equivalent 8.1 servicegen element	Attribute
messageservice	name	No equivalent.	No equivalent.
	destination	servicegen.service	JMSDestination
	destinationtype	servicegen.service	JMSDestinationType
	action	servicegen.service	JMSAction
	connectionfactory	servicegen.service	JMSConnectionFactory
	uri	servicegen.service	serviceURI
clientjar	path	servicegen.service.client	clientJarName

Updating the URL Used to Access the Web Service

The default URL used by client applications to access a WebLogic Web Service and its WSDL has changed between versions 6.1 and 8.1 of WebLogic Server.

In Version 6.1, the default URL was:

`[`*`protocol`*`]://[`*`host`*`]:[`*`port`*`]/[`*`context`*`]/[`*`WSname`*`]/[`*`WSname`*`].wsdl`

as described in *URLs to Invoke WebLogic Web Services and Get the WSDL* at http://e-docs.bea.com/wls/docs61/webServices/client.html#client008.

For example, the URL to invoke a 6.1 Web Service built with the `build.xml` file shown in "Converting a 6.1 build.xml file to 8.1" on page 21-3, is:

```
http://host:port/weather/statelessSession.WeatherHome/statelessSession.Weather
Home.wsdl
```

In 8.1, the default URL is:

`[`*`protocol`*`]://[`*`host`*`]:[`*`port`*`]/[`*`contextURI`*`]/[`*`serviceURI`*`]?WSDL`

as described in "WebLogic Web Services Home Page and WSDL URLs" on page 6-21.

For example, the URL to invoke the equivalent 8.1 Web Service after converting the 6.1 `build.xml` file shown in "Converting a 6.1 build.xml file to 8.1" on page 21-3 and running `wsgen` is:

```
http://host:port/weather/weatheruri?WSDL
```

CHAPTER 22

Using WebLogic Workshop With WebLogic Web Services

The following sections describe how to use WebLogic Workshop to create WebLogic Web Services:

- "Overview of WebLogic Workshop and WebLogic Web Services" on page 22-1
- "Using WebLogic Workshop To Create a WebLogic Web Service: A Simple Example" on page 22-4
- "Using WebLogic Workshop To Create a WebLogic Web Service: A More Complex Example" on page 22-7

Overview of WebLogic Workshop and WebLogic Web Services

This document provides examples and scenarios of using different technologies of WebLogic Platform (Workshop IDE) to create WebLogic Web Services. The overview information is divided into the following topics:

- "WebLogic Workshop and WebLogic Web Services" on page 22-1
- "EJBGen" on page 22-2
- "Using Meta-Data Tags When Creating EJBs and Web Services" on page 22-3

WebLogic Workshop and WebLogic Web Services

Today in Java, there are two main programming models for developing Web Services. Both of these models are supported by BEA. The first is defined in JAX-RPC, which relies on a back-end

EJB or a plain Java Object to provide the business logic of the Web Service. To develop Web Services using this model you can use the WebLogic Web Services Ant tasks (such as `servicegen`). The second model is based on code annotation as defined in JSR-181, Web Services Metadata for the Java Platform. The model used to develop Web Services in WebLogic Workshop is a precursor to JSR-181. Whether you decide to use the JAX-RPC model or the WebLogic Workshop model, the Web Services you develop will deploy and run on WebLogic Server, however the SOAP implementation and dispatch model will vary depending on which programming model you choose.

Typically, this difference in SOAP implementation is transparent and unimportant. However, due to the differences in the programming models and the slight differences in the characteristics of the two runtimes, you sometimes might want to use WebLogic Workshop to create applications that run on the runtime supported by the JAX-RPC programming model. To do so, you cannot use annotated JWS files, the standard way to create Web Services in WebLogic Workshop. Rather, you use WebLogic Workshop to create the back-end component (stateless session EJB), export the EJB and an Ant build script that builds the component, then add calls to the Web Service Ant tasks to the build script to package everything up into a Web Service than runs on the runtime supported by the JAX-RPC model. The examples in this document show how to go through this process.

Once you have exported the EJB to a JAR file from WebLogic Workshop, follow the standard guidelines outlined in this book to create a Web Service that runs on the runtime supported by the JAX-RPC programming model. In particular, refer to:

- Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks" for procedural information about using the Ant tasks.
- Appendix B, "Web Service Ant Tasks and Command-Line Utilities" for reference information about the Ant tasks.
- "Deploying and Testing WebLogic Web Services" on page 6-21 for information about deploying and testing the Web Service.

EJBGen

When you use WebLogic Workshop to create a stateless session EJB back-end component, you are actually using a plug-in called EJBGen, an EJB 2.0 code generator. When you write the code for your EJB in Workshop, you use special `@ejbgen` Javadoc tags to describe what the EJB looks like, and then, when you build your EJB, Workshop calls the EJBGen plug-in to generate the remote and home interface classes and the deployment descriptor files.

EJBGen can also be executed as a command-line utility, which means that the same `*.ejb` file you use in Workshop can also be processed outside of Workshop. (The only difference is that you must change the extension from `*.ejb` to `*.java`.) Use the java command, as shown in the following example:

```
java weblogic.tools.ejbgen.EJBGen myEJB.java
```

One way of using the command-line version of EJBGen is to add it to the Ant build script that calls the Ant tasks, such as `servicegen`, to build your WebLogic Web Service so that you can re-generate your EJB without having to use Workshop.

Because you can use EJBGen both within Workhop and as a command-line utility, it is assumed in the examples in this document that you might use either flavor, even if the example describes one particular flavor.

For details about using the command-line EJBGen tool, see EJBGen Reference at http://e-docs.bea.com/wls/docs81/ejb/EJBGen_reference.html. For details about the EJBGen Workshop plug-in, see the *Developing Enterprise JavaBeans* topic in the left frame of the WebLogic Workshop Help at http://e-docs.bea.com/wls/docs81/../../workshop/docs81/doc/en/core/index.html.

Using Meta-Data Tags When Creating EJBs and Web Services

The examples in this document show how to use meta-data tags in Java source code files to create Web Services. These meta-data tags come in two flavors:

- those used by EJBgen, specified with the `@ejbgen` Javadoc tag.
- those used by the WebLogic Web Service `source2wsdd` Ant task, specified with the `@wlws` Javadoc tag.

You use meta-data Javadoc tags in a Java source file to specify in more detail what an EJB, and the Web Service that exposes the EJB, look like. Then you use either EJBGen or the `source2wsdd` Ant task (or both) to generate the additional components. In particular, EJBGen generates the EJB deployment descriptors and the EJB Home and Remote interfaces; the `source2wsdd` Ant task generates the Web Services deployment descriptor file.

For reference information about the EJBGen tags, see EJBGen Reference at http://e-docs.bea.com/wls/docs81/ejb/EJBGen_reference.html. For information about the `source2wsdd` tags, see Appendix C, “source2wsdd Tag Reference.”

Using WebLogic Workshop To Create a WebLogic Web Service: A Simple Example

This section describes a simple example of creating a WebLogic Web Service using the WebLogic Workshop IDE.

Note: This procedure works only with Service Pack 2 of WebLogic Workshop. For an example that works on the GA version of WebLogic Workshop, see "Using WebLogic Workshop To Create a WebLogic Web Service: A More Complex Example" on page 22-7.

The example first uses Workshop to create a stateless session EJB called `PurchaseOrderBean`, and then uses the `servicegen` WebLogic Web Services Ant task to expose the EJB as a Web Service that runs on the runtime supported by the JAX-RPC programming model. In this example, all the business logic is directly in the `PurchaseOrderBean`, which exposes two methods as Web Service operations: `submitPO` and `getStatus`.

Note: The following procedure does not always describe the exact steps you must perform in the IDE to create the various projects and objects. For this kind of detailed information, see the *Developing Enterprise JavaBeans* topic in the left frame of the WebLogic Workshop Help at http://e-docs.bea.com/wls/docs81/../../workshop/docs81/doc/en/core/index.html.

1. Invoke WebLogic Workshop from the Start menu.
2. If one does not already exist, create an application that will contain the stateless session EJB.

   It is assumed in the procedure that the application is named `myApp`.
3. If one does not already exist, create an EJB project under your Workshop application.

   It is assumed that the project is named `myEJBs`.
4. Create a folder under the EJB project.

   It is assumed that the folder is named `myPackage`.
5. Create a Session bean under the `myPackage` folder.

   It is assumed that your EJB is named `PurchaseOrderBean.ejb`.
6. Click Source View and update `PurchaseOrderBean.ejb`, replacing all the code after the `package myPackage` statement with the following code:

```
import javax.ejb.*;
import weblogic.ejb.*;
```

```
/**
* @ejbgen:session
*      ejb-name = "PurchaseOrder"
*
* @ejbgen:jndi-name
*      remote = "ejb.PurchaseOrderRemoteHome"
*
* @ejbgen:file-generation
*      remote-class = "true"
*      remote-class-name = "PurchaseOrder"
*      remote-home = "true"
*      remote-home-name = "PurchaseOrderHome"
*      local-class = "false"
*      local-class-name = "PurchaseOrderLocal"
*      local-home = "false"
*      local-home-name = "PurchaseOrderLocalHome"
 */
public class PurchaseOrderBean
  extends GenericSessionBean
  implements SessionBean
{
  public void ejbCreate() {
    // Your code here
  }

    /**
     * @ejbgen:remote-method
     */
    public long submitPO(String PoText)
    {
        return System.currentTimeMillis();
    }

    /**
     * @ejbgen:remote-method
     */
    public int getStatus(long orderNo)
    {
        return 0;
    }
}
```

7. Ensure that the EJB builds correctly by right-clicking on the `myEJBs` project in the application pane and selecting *Build myEJBs*.

8. Export the Ant build file that builds the `PurchaseOrder` EJB outside of Workshop by clicking Tools->Application Properties..., choosing Build in the left pane, and clicking the Export to Ant File button.

A file called `exported_build.xml` is generated in the project directory of your Workshop application.

Make a note of the project directory, listed under the EAR heading in the right pane.

9. Open a command window change to the project directory of your Workshop application.
10. Edit the `exported_build.xml` file, adding the following elements to invoke the `servicegen` Ant task on the `PurchaseOrder` EJB you created in Workshop:
    - A taskdef definition for the `servicegen` Ant task:

```
  <taskdef name="servicegen"

classname="weblogic.ant.taskdefs.webservices.servicegen.ServiceGenTask"
          classpath="${server.classpath}" />
```

    - A target for the `servicegen` Ant task:

```
     <target name="servicegen">
       <delete file="${output.file}" />
       <servicegen
         destEar="${output.file}"
       >
         <service
            ejbJar="myEJBs.jar"
            serviceName="PurchaseOrderService"
            serviceURI="/PurchaseOrderService"
            targetNamespace="http://example.com/PurchaseOrderService"
         />
       </servicegen>
     </target>
```

11. Set your environment by executing the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.
12. Execute the `build` and `servicegen` Ant tasks of the `exported_build.xml` file:

    ```
    prompt> ant -f exported_build.xml build servicegen
    ```

    The `servicegen` target of the Ant task updates the application, exposing the `PurchaseOrder` EJB as a WebLogic Web Service.

13. Deploy the application as usual. For details, see "Deploying and Testing WebLogic Web Services" on page 6-21.

Using WebLogic Workshop To Create a WebLogic Web Service: A More Complex Example

This section describes a more complex example of creating a WebLogic Web Service using the WebLogic Workshop IDE.

Description of the Example

In the example, the `PurchaseOrderServiceBean` EJB is exposed as a Web Service, but it does not contain any business logic. It has one operation that accepts a purchase order number from an incoming SOAP request and returns a `PurchaseOrder` object in the SOAP response. The `PurchaseOrderServiceBean` EJB that is exposed as a Web Service delegates all the actual work of looking up a purchase order to a conventional session facade EJB called `PurchasingManagerBean`. This EJB implements all the business logic of the application, independent of the Web Service entry point. The EJB uses the `Item` and `PurchaseOrder` complex data types when processing purchase orders, and creates new `PurchaseOrder` objects using the `PurchaseOrderFactory`.

The `PurchaseOrderServiceBean` EJB, in addition to using EJBGen Javadoc tags as in the preceding example, also uses WebLogic Web Service `source2wsdd` tags, identified with the `@wlws` prefix. Because of the use of meta-data tags, the example uses individual Ant tasks, such as `source2wsdd` and `autotype`, to assemble a Web Service, rather than the all-encompassing `servicegen` Ant task. For details about the `source2wsdd` tags, see Appendix C, "source2wsdd Tag Reference."

The example also shows how to use a SOAP message handler. The SOAP message handler looks for a SOAP header called `My-Username` in the SOAP request from a client, and if it exists, it extracts the value, and logs a message that includes the name to a log file. If the header does not exist, the SOAP message handler logs a message with `Unknown` as the username. For additional information about SOAP message handlers, see Chapter 12, "Creating SOAP Message Handlers to Intercept the SOAP Message."

Assumptions

This main point of this example is to show how to use WebLogic Workshop to create an EJB and SOAP message handler that together will be exposed as a Web Service, and then how to package

it all together into a deployable EAR file than runs on the runtime supported by the JAX-RPC programming model. For this reason, it is assumed that you have already:

- Created an application in WebLogic Workshop, called `myComplexApp`, that contains an EJB Project called `PurchaseOrderService`.
- Within the `PurchaseOrderService` EJB project, created a folder called `po` that contains the EJB that performs all the business logic (`PurchasingManagerBean`) as well as the various objects used by the `PurchasingManagerBean` EJB, such as `Item`, `PurchaseOrder`, and `PurchaseOrderFactory`.

  See "Source Code for Supporting Java Objects" on page 22-16 for sample source code for these objects.

The Example

Note: The following procedure does not always describe the exact steps you must perform in the IDE to create the various projects and objects. For this kind of detailed information, see the *Developing Enterprise JavaBeans* topic in the left frame of the WebLogic Workshop Help at http://e-docs.bea.com/wls/docs81/../../workshop/docs81/doc/en/core/index.html.

1. Invoke WebLogic Workshop from the Start menu.
2. Create a folder under the `PurchaseOrderService` EJB project called `service`.
3. Create a Session bean under the `service` folder called `PurchaseOrderServiceBean.ejb`.
4. Click Source View and update `PurchaseOrderBean.ejb`, replacing all the Workshop-generated Java code with the following code:

```
package service;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import po.PurchasingManagerLocal;
import po.PurchasingManagerLocalHome;

/**
 * This is the web service facade. It defines the web service
 * operations and contains any web service-specific
```

```
 * application logic (such as logging invokes in a handler).
 *
 * @ejbgen:session
 *   ejb-name = "PurchaseOrderServiceEJB"
 * @ejbgen:jndi-name
 *   local = "PurchaseOrderService"
 * @ejbgen:ejb-local-ref
 *   link = "PurchasingManagerEJB"
 * @wlws:webservice
 *  name="PurchaseOrderService"
 *  targetNamespace="http://openuri.org/easypo_service"
 *  style="document"
 */

public class PurchaseOrderServiceBean implements SessionBean {

     // local interface of the PurchasingManager session facade
     PurchasingManagerLocal pm = null;

    /**
     * This operation return a PurchaseOrder that is retrieved from
     *  the PurchasingManager

     *  @ejbgen:local-method
     *  @wlws:operation handler-chain = "PurchaseOrderServiceHandlerChain"
     */
    public po.PurchaseOrder getPurchaseOrder(String poNumber) {
        return pm.getPO(poNumber);
    }
    public void ejbCreate() throws CreateException {
        try {
            InitialContext ctx = new InitialContext();
            PurchasingManagerLocalHome pmhome = (PurchasingManagerLocalHome)
ctx.lookup("java:/comp/env/ejb/PurchasingManagerEJB");
            pm = pmhome.create();
        } catch (NamingException ne) {
           throw new CreateException("Could not locate PurchasingManager EJB");
        }
    }

    public void ejbRemove() {
    }

    public void ejbPassivate() {
    }

    public void ejbActivate() {
    }
```

```
    public void setSessionContext(SessionContext ctx) {
    }

}
```

5. Create a Java class under the `service` folder called `PurchaseOrderServiceHandler.java`. This class implements the server-side SOAP message handler.

6. In the middle pane, replace all the Workshop-generated Java code of `PurchaseOrderServiceHandler.java` with the following code:

```
package service;

import javax.xml.rpc.handler.Handler;
import javax.xml.rpc.handler.MessageContext;
import javax.xml.rpc.handler.HandlerInfo;
import javax.xml.rpc.handler.soap.SOAPMessageContext;
import javax.xml.namespace.QName;
import javax.xml.soap.*;
import java.io.FileWriter;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Map;
import java.util.Iterator;
import java.util.Date;

/**
 * Represents a JAX-RPC handler that intercepts an incoming SOAP message,
 * extracts an optional header called"My-Username" and logs a message to a
 * log file.
 */

public class PurchaseOrderServiceHandler implements Handler {
    private static final String HEADER_NAME = "My-Username";
    private PrintWriter out;

    public QName[] getHeaders() {
        return new QName[]{new QName(HEADER_NAME)};
    }

    public boolean handleRequest(MessageContext messageContext) {
        String userName = null;
        try {
            userName = getHeaderValue(messageContext, HEADER_NAME);
        } catch (SOAPException e) {
            throw new RuntimeException("Could not retrieve header value", e);
        }
        if (userName == null) userName = "UNKNOWN";
        out.println(new Date() + ": Received request from username " + userName);
```

```
        out.flush();
        return true;
    }

    public boolean handleResponse(MessageContext messageContext) {
        return true;
    }

    public boolean handleFault(MessageContext messageContext) {
        return true;
    }

    public void init(HandlerInfo handlerInfo) {
        Map config = handlerInfo.getHandlerConfig();
        String logFileName = (String) config.get("logFile");
        if (logFileName == null) {
            throw new RuntimeException("Could not initialize handler; logFile
not specified in init-params.");
        }
        try {
            out = new PrintWriter(new FileWriter(logFileName));
        } catch (IOException e) {
           throw new RuntimeException("Could not initialize handler; could not
open log file " + logFileName, e);
        }
    }

    public void destroy() {
        out.close();
    }

    private static String getHeaderValue(MessageContext messageContext,
             String headerName) throws SOAPException {
        SOAPFactory fact = SOAPFactory.newInstance();
        SOAPMessageContext ctx = (SOAPMessageContext) messageContext;
        SOAPHeader headers =
             ctx.getMessage().getSOAPPart().getEnvelope().getHeader();

        Iterator i = headers.getChildElements(fact.createName(HEADER_NAME));
        while (i.hasNext()) {
            SOAPElement elt = (SOAPElement) i.next();
            if (headerName.equals(elt.getElementName().getLocalName())) {
                return elt.getValue();
            }
        }
        return null;
    }
}
```

7. Create an XML file called `handler-chain.xml` under the `service` folder.

This XML file will contain a description of the SOAP message handler and handler chain used by the Web Service. The file will late be used by the `source2wsdd` Ant task when generating the Web Service deployment descriptor.

8. In the middle pane, replace the Workshop-generated `<root></root>` XML elements with the following XML:

```
<handler-chains>
    <handler-chain name="PurchaseOrderServiceHandlerChain">
        <handler class-name="service.PurchaseOrderServiceHandler">
            <init-params>
                <init-param name="logFile" value="./posvc.log" />
            </init-params>
        </handler>
    </handler-chain>
</handler-chains>
```

9. Click Tools->Application Properties... and choose Build in the left pane. Make a note of the project directory, listed under the EAR heading in the right pane.

10. Open a command window and change to the project directory of your Workshop application.

11. Set your environment by executing the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\`*`domainName`*, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain

12. Ensure you can rebuild your EJB project from the command line by exeucting the *`WL_HOME`*`\workshop\wlwBuild.cmd`, where *`WL_HOME`* refers to the main WebLogic Platform directory, such as `c:\beahome\weblogic81`. Use the `-project` option to pass it the name of the EJB project, as shown in the following example:

    ```
    prompt> c:\beahome\weblogic81\workshop\wlwBuild.cmd -project
    PurchaseOrderService
    ```

    You can add this line to your automated shell scripts that iteratively build the application.

13. In the Workshop project directory, create an Ant `build.xml` file that includes calls to the `autotype` and `source2wsdd` WebLogic Web Service ant tasks.

    These Ant tasks take the compiled EJB and SOAP message handler class and create the needed Web Services components, such as the deployment descriptor and data type components. For an example of this file, see "Sample build.xml File" on page 22-13.

14. Execute the Ant tasks by running the `ant` command:

```
prompt> ant
```

If you use a build.xml file similar to the sample, the Ant task create a deployable EAR file called `PurchaseOrderService.ear` in a directory called output that is parallel to the Workshop project directory.

15. Deploy the `PurchaseOrderService.ear` file as usual. For details, see "Deploying and Testing WebLogic Web Services" on page 6-21.

Sample build.xml File

```
<project name="build-scenario1" default="build">

    <!-- WebLogic Home. -->
    <property name="platformhome" value="/home/toddk/bea/weblogic81"/>

    <!-- base url of the server and administrator user, password -->
    <property name="server_url" value="http://localhost:7001"/>
    <property name="admin_user" value="weblogic"/>
    <property name="admin_passwd" value="gumby1234"/>

    <!-- location of the browser executable -->
    <property name="browser"
              value="/usr/local/MozillaFirebird/MozillaFirebird"/>

    <!-- the dir into which the output of compilers and tools is directed -->
    <property name="output_dir" value="../output"/>

   <!-- the name of service (used in constructing war file name, WSDL, etc.) -->
    <property name="service_name" value="PurchaseOrderService"/>

    <!-- the Java package in which the service class is located -->
    <property name="service_package" value="service"/>

    <!-- the target namespace of the service -->
    <property name="target_namespace"
              value="http://openuri.org/easypo_service"/>

    <!-- name of the Workshop EJB project containing the web service EJBs -->
    <property name="service_ejb_project" value="PurchaseOrderService" />

   <!-- the dir that contains the exploded ear file containing the web service
-->
    <property name="output_ear" value="${output_dir}/${service_name}-ear"/>
    <path id="build.classpath">
        <pathelement path="${java.class.path}"/>
        <pathelement location="${platformhome}/server/lib/webservices.jar"/>
        <pathelement location="${output_ear}"/>
        <pathelement location="${output_ear}/APP-INF/classes"/>
```

```
        <pathelement location="${service_ejb_project}.jar"/>
    </path>
    <target name="build"
  depends="clean, setup, webservice.build, webservice.client, build.finish"/>
    <target name="clean" description="delete generated stuff">
        <delete dir="${output_dir}"/>
    </target>

    <target name="setup" description="create output directories">
        <mkdir dir="${output_ear}/META-INF"/>
        <mkdir dir="${output_dir}/${service_name}-war/WEB-INF"/>
    </target>

    <!-- build the web service from the web service EJB JAR -->
    <target name="webservice.build">

        <!-- put the EJB JAR in the exploded EAR -->
        <copy file="${service_ejb_project}.jar" todir="${output_ear}" />

        <!-- generate XML types from the Java value types in the service -->
        <autotype javaComponents="${service_package}.${service_name}Local"
            typeMappingFile="${output_ear}/APP-INF/classes/types.xml"
            destDir="${output_ear}/APP-INF/classes"
            packageName="${service_package}"
            classpathref="build.classpath"/>

        <!-- build the service from the EJB and autotyper types -->
        <source2wsdd

javaSource="${service_ejb_project}/${service_package}/${service_name}Bean.ejb"
           ddFile="${output_dir}/${service_name}-war/WEB-INF/web-services.xml"
            typesInfo="${output_ear}/APP-INF/classes/types.xml"

handlerInfo="${service_ejb_project}/${service_package}/handler-chain.xml"
            serviceURI="/${service_name}"
            wsdlFile="${output_dir}/${service_name}-war/${service_name}.wsdl"
            ejblink="${service_ejb_project}.jar#${service_name}EJB"
            classpathref="build.classpath">
        </source2wsdd>

        <!-- package the web service war -->
        <jar destFile="${output_ear}/${service_name}.war"
basedir="${output_dir}/${service_name}-war"/>

        <!-- package the ear -->
        <echo file="${output_ear}/META-INF/application.xml">
            <![CDATA[
            <!DOCTYPE application PUBLIC '-//Sun Microsystems, Inc.//DTD J2EE
Application 1.2//EN'
                'http://java.sun.com/j2ee/dtds/application_1_2.dtd'>
```

```
            <application>
              <display-name> ${service_name}-Service </display-name>
              <module>
                <ejb>
                   ${service_ejb_project}.jar
                </ejb>
              </module>
              <module>
                <web>
                   <web-uri> ${service_name}.war </web-uri>
                   <context-root> ${service_name} </context-root>
                </web>
              </module>
            </application>
            ]]>
        </echo>
        <jar destFile="${output_dir}/${service_name}.ear"
baseDir="${output_ear}"/>

    </target>

    <target name="webservice.client">

        <!-- generate JAX-RPC client interfaces, stubs into the client jar -->
        <clientgen
            description="create a web service client from the ear"
            clientJar="${output_dir}/${service_name}-client"
            wsdl="${output_dir}/${service_name}-war/${service_name}.wsdl"
            typeMappingFile="${output_ear}/APP-INF/classes/types.xml"
            packageName="${service_package}.client"
            usePortNameAsMethodName="true"
            keepgenerated="true"
            classpathref="build.classpath">
        </clientgen>
        <!-- compile the client app into the client jar -->
        <javac srcdir="."
            includes="client/*.java"
            destdir="${output_dir}/${service_name}-client"
            classpathref="build.classpath">
        </javac>

        <!-- package the client jar; executing the jar runs the test app -->
        <jar
            destFile="${output_dir}/${service_name}-client.jar"
            baseDir="${output_dir}/${service_name}-client"/>

    </target>

    <target name="build.finish" description="clean up temp files">
        <delete dir="${output_dir}/${service_name}-client"/>
```

```
        <delete dir="${output_dir}/${service_name}-war"/>
        <delete dir="${output_ear}"/>
    </target>

    <target name="deploy" description="deploy service">
        <wldeploy action="deploy" source="${output_dir}/${service_name}.ear"
            adminurl="${server_url}"
            user="${admin_user}" password="${admin_passwd}"/>
    </target>

    <target name="undeploy" description="undeploy service">
        <wldeploy action="undeploy" source="${output_dir}/${service_name}.ear"
            adminurl="${server_url}"
            user="${admin_user}" password="${admin_passwd}"/>
    </target>

    <target name="browse" description="browse test page for this service">
        <exec executable="${browser}">
            <arg line="${server_url}/${service_name}/${service_name}"/>
        </exec>
    </target>

    <target name="run" description="run client">
        <java classname="client.Main" fork="true">
            <classpath>
                <pathelement path="${java.class.path}"/>
             <pathelement location="${output_dir}/${service_name}-client.jar"/>

            </classpath>
            <arg line="${server_url}/${service_name}/${service_name}?WSDL"/>
            <jvmarg line="-Dweblogic.webservice.verbose=true"/>
        </java>
    </target>

</project>
```

Source Code for Supporting Java Objects

This section provides sample code for the following supporting Java objects which are already assumed to exist:

- Item.java
- PurchaseOrder.java
- PurchasingManagerBean.java
- PurchaseOrderFactory.java

Item.java

```
package po;

/**
 * Represents a single item on a purchase order.
 */
public class Item {
    private String catNumber;
    private String description;
    private int quantity;

    public Item() {
    }

    public Item(String catNumber, String description, int quantity) {
        this.catNumber = catNumber;
        this.description = description;
        this.quantity = quantity;
    }

    public String getCatNumber() {
        return catNumber;
    }

    public void setCatNumber(String catNumber) {
        this.catNumber = catNumber;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public int getQuantity() {
        return quantity;
    }

    public void setQuantity(int quantity) {
        this.quantity = quantity;
    }
```

```
    public String toString() {
        StringBuffer sbuf = new StringBuffer();
        sbuf.append("[Item");
        sbuf.append("\n\tcatNumber = " + catNumber);
        sbuf.append("\n\tdescription = " + description);
        sbuf.append("\n\tquantity = " + quantity);
        sbuf.append("\n]");
        return sbuf.toString();
    }
}
```

PurchaseOrder.java

```
package po;

import po.Item;

/**
 * Date: Oct 15, 2003
 * Time: 3:29:23 PM
 */
public class PurchaseOrder {
    private String poNumber;
    private Item[] items;
    private String custName;
    private String custAddress;

    public PurchaseOrder() {
    }

    public PurchaseOrder(String poNumber) {
        this.poNumber = poNumber;
    }

    public String getPoNumber() {
        return poNumber;
    }

    public void setPoNumber(String poNumber) {
        this.poNumber = poNumber;
    }
```

```
    public Item[] getItems() {
        return items;
    }
    public void setItems(Item[] items) {
        this.items = items;
    }
    public String getCustName() {
        return custName;
    }
    public void setCustName(String custName) {
        this.custName = custName;
    }
    public String getCustAddress() {
        return custAddress;
    }
    public void setCustAddress(String custAddress) {
        this.custAddress = custAddress;
    }
    public String toString() {
        StringBuffer sbuf = new StringBuffer();
        sbuf.append("[PurchaseOrder");
        sbuf.append("\n\tpoNumber = " + poNumber);
        sbuf.append("\n\tcustName = " + custName);
        sbuf.append("\n\tcustAddress = " + custAddress);
        if (items != null) {
            for (int i = 0; i < items.length; ++i) {
                sbuf.append("\n");
                sbuf.append(items[i].toString());
            }
        }
        sbuf.append("\n]");
        return sbuf.toString();
    }
}
```

PurchasingManagerBean.java

```
package po;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

/**
 * This is a session facade EJB that is the entry point to the business
 * logic ofthe application.
 *
 * @ejbgen:session
 *   ejb-name = "PurchasingManagerEJB"
 * @ejbgen:jndi-name
 *   local = "PurchasingManager"
 */
public class PurchasingManagerBean implements SessionBean {

    /**
     * @ejbgen:local-method
     */
    public PurchaseOrder getPO(String poNumber) {
       return PurchaseOrderFactory.createPO();  // always return same thing
    }

    /**
     * @ejbgen:local-method
     */
    public int getStatus(String poNumber) {
        return 1;
    }

    public void ejbRemove() {}
    public void ejbCreate() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void setSessionContext(SessionContext ctx) {}

}
```

PurchaseOrderFactory.java

```
package po;
```

```
import java.util.ArrayList;
import java.util.List;

/**
 * A Factory to create PurchaseOrders. This just creates the same
 * dummy PO each time.
 */
public class PurchaseOrderFactory {

    /**
     * Constructs a PurchaseOrder object
     */
    public static PurchaseOrder createPO() {

        PurchaseOrder po = new PurchaseOrder("PO8048392");

        // Add customer
        po.setCustName("Mary Mary Quite Contrary");
        po.setCustAddress("123 Main Street, Hogsmeade");

        // Add Line Items
        List items = new ArrayList();
        items.add(new Item("S-123", "Lacewing Flies", 100));
        items.add(new Item("S-456", "Leeches", 3));
        items.add(new Item("S-043", "Powdered Bicon Horn", 1));
        items.add(new Item("S-153", "Knotgrass", 5));
        items.add(new Item("S-904", "Fluxweed", 1));
        items.add(new Item("S-034", "Boomslang Skin", 2));
        po.setItems((Item[]) items.toArray(new Item[]{}));

        return po;
    }
}
```

APPENDIX A

WebLogic Web Service Deployment Descriptor Elements

The following sections describe the `web-services.xml` file using different formats:

- "Overview of web-services.xml" on page A-1
- "Graphical Representation" on page A-1
- "Element Reference" on page A-4

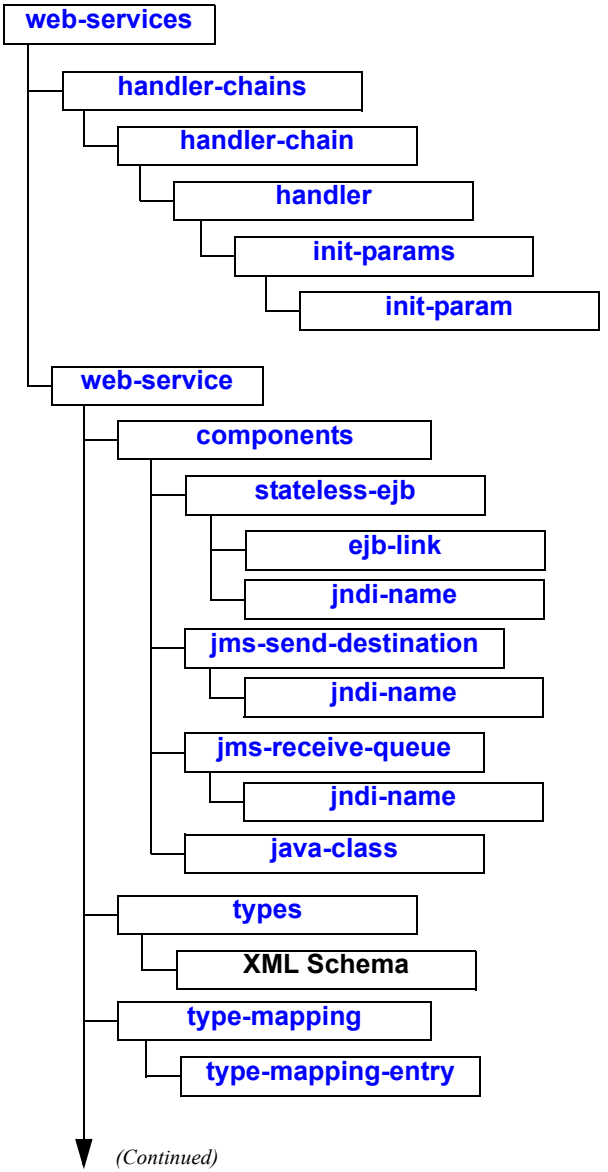
Overview of web-services.xml

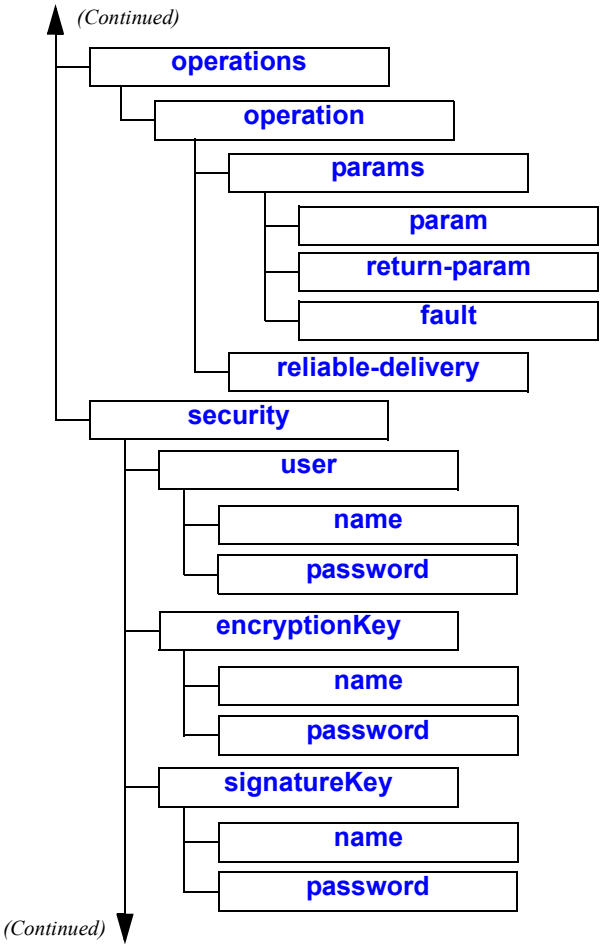
The `web-services.xml` deployment descriptor file contains information that describes one or more WebLogic Web Services. This information includes details about the back-end components that implement the operations of a Web Service, the non-built-in data types used as parameters and return values, the SOAP message handlers that intercept SOAP messages, and so on. As is true for all deployment descriptors, `web-services.xml` is an XML file.

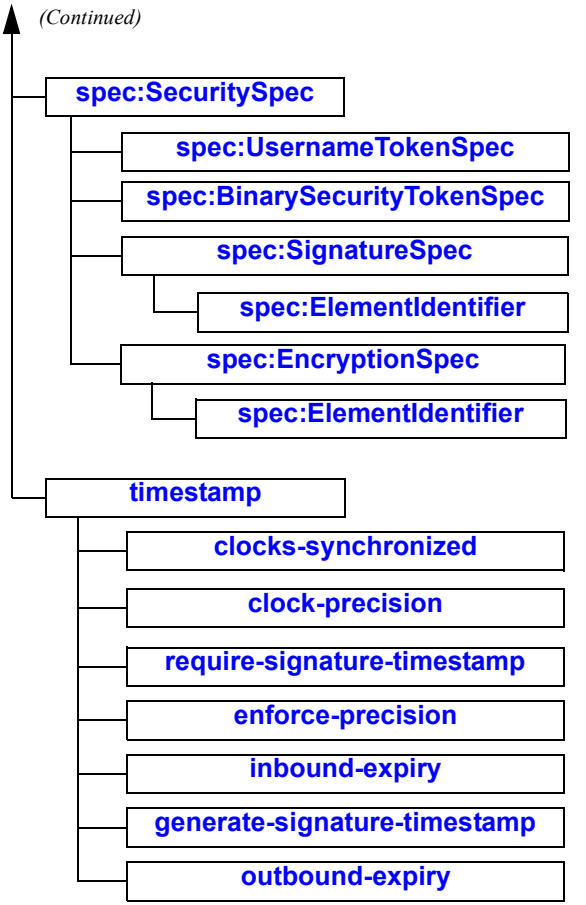
Graphical Representation

The following graphic describes the `web-services.xml` element hierarchy.

Figure 22-1 web-services.xml Element Hierarchy




(Continued)
operations
operation
params
param
return-param
fault
reliable-delivery
security
user
name
password
encryptionKey
name
password
signatureKey
name
password
(Continued)



Element Reference

The following sections, arranged alphabetically, describe each element in the `web-services.xml` file.

See "Examining Different Types of web-services.xml Files" on page E-9 for sample Web Services deployment descriptor files for a variety of different types of WebLogic Web Services.

clock-precision

Describes the accuracy of synchronization between the clock of a client application invoking a WebLogic Web Service and WebLogic Server's clock. WebLogic Server uses this value to account for a reasonable level of clock skew between two clocks.

The value is expressed in milliseconds. This means, for example, that if the clocks are accurate within a one minute of each other, the value of this element is 60000.

If the value of this element is greater than the expiration period of an incoming SOAP request, WebLogic Server rejects the request because it cannot accurately enforce the expiration. For example, if the clock precision value is 60000 milliseconds, and WebLogic Server receives a SOAP request that expires 30000 milliseconds after its creation time, it is possible that the message has lived for longer than 30000 seconds, due to the 60000 millisecond clock precision discrepancy, so WebLogic Server has no option but to reject the message. You can relax this strict enforcement by setting the `<enforce-precision>` element to `false`. For details, see "enforce-precision" on page A-7.

This element must be specified in conjunction with `<clocks-synchronized>`.

The default value for this element is 60000.

This element does not have any attributes. This element is a child element of `<timestamp>`.

clocks-synchronized

Specifies whether WebLogic Server assumes that the clocks of the client application invoking a WebLogic Web Service and WebLogic Server are synchronized when dealing with timestamps in SOAP messages.

If the value of this element is `true`, WebLogic Server enforces, if it exists, the time expiration of the SOAP request from a client application that is invoking a WebLogic Web Service. This means that if the time stamp has expired, WebLogic Server does not invoke the Web Service operation. If the value of this element is `false`, WebLogic Server rejects all SOAP requests that contain a time expiration.

Valid values for this element are `true` and `false`. The default value is `false`.

This element does not have any attributes. This element is a child element of `<timestamp>`.

components

Defines the back-end components that implement the Web Service.

A WebLogic Web Service can be implemented using one or more of the following components:

- Stateless session EJB
- JMS destination
- A Java class

This element has no attributes.

ejb-link

Identifies which EJB in an EJB JAR file is used to implement the stateless session EJB back-end component.

Table A-1 Attributes of the <ejb-link> Element

Attribute	Description	Datatype	Required?
path	Name of the EJB in the form of: *jar-name*#*ejb-name* *jar-name* refers to the name of the JAR file, contained within the Web Service EAR file, that contains the stateless session EJB. The name should include pathnames relative to the top level of the EAR file. *ejb-name* refers to the name of the stateless session EJB, corresponding to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor file in the EJB JAR file. Example: `myapp.jar#StockQuoteBean`	String	Yes

encryptionKey

Specifies the name and password of a key pair and certificate used when encrypting elements of the SOAP message. Specify the name using the `<name>` subelement; specify the password with the `<password>` subelement.

Note: Create the key and certificate pair in the WebLogic Server keystore with the Administration Console. For details, see Storing Private Keys, Digital Certificates, and Trusted CAs.

This element does not have any attributes.

enforce-precision

Specifies whether to enforce the clock precision time period.

If this element is set to `false`, WebLogic Server does *not* reject SOAP requests whose time expiration period is smaller than the clock precision time, specified with the `<clock-precision>` element. By default, WebLogic Server rejects these SOAP requests because it cannot accurately determine whether the message has expired, due to the discrepancy in clock precision between the client application and WebLogic Server.

Valid values for this element are `true` and `false`. The default value is `true`.

This element does not have any attributes. This element is a child element of `<timestamp>`.

fault

Specifies the SOAP fault that should be thrown if there is an error invoking this operation.

This element is not required.

Table A-2 Attributes of the <fault> Element

Attribute	Description	Datatype	Required?
class-name	Fully qualified Java class that implements the SOAP fault.	String	Yes
name	Name of the fault.	String	Yes

generate-signature-timestamp

Specifies whether WebLogic Server includes a timestamp in the SOAP response to a client application that has invoked a WebLogic Web Service operation.

Valid values for this element are `true` and `false`. The default value is `true`.

This element does not have any attributes. This element is a child element of `<timestamp>`.

handler

Describes a SOAP message handler in a handler chain. A single handler chain can consist of one or more handlers.

If the Java class that implements the handler expects initialization parameters, specify them using the optional `<init-params>` child element of the `<handler>` element.

Table A-3 Attributes of the <handler> Element

Attribute	Description	Datatype	Required?
class-name	Fully qualified Java class that implements the SOAP message handler.	String	Yes

handler-chain

Lists the SOAP message handlers that make up a particular handler chain. A single WebLogic Web Service can define zero or more handler chains.

The order in which the handlers (defined by the `<handler>` child element) are listed is important. By default, the `handleRequest()` methods of the handlers execute in the order that they are listed as child elements of the `<handler-chain>` element. The `handleResponse()` methods of the handlers execute in the *reverse* order that they are listed.

Table A-4 Attributes of the <handler-chain> Elementback-end

Attribute	Description	Datatype	Required?
name	Name of this handler chain.	String	Yes

handler-chains

Contains a list of `<handler-chain>` elements that describe the SOAP message handler chains used in the Web Service described by this `web-services.xml` file. A single WebLogic Web Service can define zero or more handler chains.

This element does not have any attributes.

inbound-expiry

Specifies, in milliseconds, WebLogic Server's expiration period for a SOAP request from a client application invoking a Web Service. WebLogic Server adds the value of this element to the creation date in the time stamp of the SOAP request, accounts for clock precision, then compares the result to the current time. If the result is greater than the current time, WebLogic Server rejects the invoke.

In addition to its own expiration period for SOAP requests, WebLogic Server also honors expirations in the SOAP request message itself, specified by the client application.

To specify no expiration, set this element to `-1`.

The default value of this element is `-1`.

If you set this element to a value, be sure you also specify that the clocks between WebLogic Server and client applications are synchronized by setting the `<clocks-synchronized>` element to `true`.

init-param

Specifies a name-value pair that represents one of the initialization parameters of a handler.

Table A-5 Attributes of the <init-param> Element

Attribute	Description	Datatype	Required?
name	Name of the parameter.	String	Yes
value	Value of the parameter.	String	Yes

init-params

Contains the list of initialization parameters that are passed to the Java class that implements a handler.

This element does not have any attributes.

java-class

Describes the Java class component that implements one or more operations of a Web Service.

Table A-6 Attributes of the <java-class> Element

Attribute	Description	Datatype	Required
class-name	Fully qualified name of the Java class that implements this component.	String	Yes
name	Name of this component.	String	Yes

jms-receive-queue

Specifies that one of the operations in the Web Service is mapped to a JMS queue. Use this element to describe a Web Service operation that receives data from a JMS queue.

Typically, a message producer puts a message on the specified JMS queue, and a client invoking this Web Service operation polls and receives the message.

Table A-7 Attributes of the <jms-receive-queue> Element

Attribute	Description	Datatype	Required?
connection-factory	JNDI name of the JMS Connection factory that WebLogic Server uses to create a JMS Connection object.	String	Yes
initial-context-factory	Context factory for a non-WebLogic Server JMS implementation.	String	No
name	Name of this component.	String	Yes
provider-url	URL used to connect to a non-WebLogic Server JMS implementation.	String	No

jms-send-destination

Specifies that one of the operations in the Web Service is mapped to a JMS queue. Use this element to describe a Web Service operation that sends data to the JMS queue.

Typically, a message consumer (such as a message-driven bean) consumes the message after it is sent to the JMS destination.

Table A-8 Attributes of the <jms-send-destination> Element

Attribute	Description	Datatype	Required?
connection-factory	JNDI name of the JMS Connection factory that WebLogic Server uses to create a JMS Connection object.	String	Yes
initial-context-factory	Context factory for a non-WebLogic Server JMS implementation.	String	No
name	Name of this component.	String	Yes
provider-url	URL used to connect to a non-WebLogic Server JMS implementation.	String	No

jndi-name

Specifies a reference to an object bound into a JNDI tree. The reference can be to a stateless session EJB or to a JMS destination.

Table A-9 Attributes of the <jndi-name> Element

Attribute	Description	Datatype	Required?
path	Path name to the object from the JNDI context root.	String	Yes

name

Depending on the parent element, the `<name>` element specifies:

- The username used in the username token in the SOAP response message. (Parent element is `<user>`.)
- The name of the key pair and certificate, stored in WebLogic Server's keystore, used to encrypt part of the SOAP message. (Parent element is `<encryptionKey>`.)
- The name of the key pair and certificate, stored in WebLogic Server's keystore, used to digitally sign part of the SOAP message. (Parent element is `<signatureKey>`.)

This element does not have any attributes.

operation

Configures a single operation of a Web Service. Depending on the value and combination of attributes for this element, you can configure the following types of operations:

- An invoke of a method of a stateless session EJB or Java class. Specify this type of operation by setting the `component` attribute to the name of the stateless session EJB or Java class component and the `method` attribute to the name of the method.
- An invoke of a JMS back-end component. Specify this type of operation by setting the `component` attribute to the name of the JMS component.
- The sequential invoke of the SOAP message handlers on a handler chain together with the invoke of a back-end component. Specify this type of operation by setting the `component` attribute to the name of the component, and the `handler-chain` attribute to the name of the handler chain you want to invoke.
- The sequential invoke of the SOAP message handlers on a handler chain, but with *no* back-end component. Specify this type of operation by just setting the `handler-chain` attribute to the name of the handler chain you want to invoke and *not* setting the `component` and `method` attributes.

Use the `<params>` child element to explicitly specify the parameters and return values of the operation.

Table A-10 Attributes of the <operation> Element

Attribute	Description	Datatype	Required?
component	Name of the component that implements this operation. The value of this attribute corresponds to the `name` attribute of the appropriate `<component>` element.	String	No
handler-chain	Name of the SOAP message handler chain that implements the operation. If you specify this attribute along with the `component` and `method` attributes, then the operation is implement with both the method and the handler chain. If, however, you do *not* specify the `component` and `method` attributes, but rather specify `handler-chain` on its own, then the operation is implemented with just a SOAP message handler chain. The value of this attribute corresponds to the `name` attribute of the appropriate `<handler-chain>` element.	String	No
in-security-spec	Specifies the name of the security specification that describes the message-level security of the client application's SOAP request when it invokes the operation. The security specification describes what part of the SOAP request should be encrypted or digitally signed. If you do not specify this attribute, the default security specification, if it exists, is applied to the SOAP request. If there is no default security specification, then no message-level security is applied to the SOAP request. The value of this attribute corresponds to the `Id` attribute of the appropriate `<spec:SecuritySpec>` element.	String	No.

Table A-10 Attributes of the <operation> Element

Attribute	Description	Datatype	Required?
invocation-style	Specifies whether the operation both receives a SOAP request and sends a SOAP response, or whether the operation only receives a SOAP request but does *not* send back a SOAP response. This attribute accepts only two values: `request-response` (default value) or `one-way`. **Note:** If the back-end component that implements this operation is a method of a stateless session EJB or Java class and you set this attribute to `one-way`, the method must return `void`.	String	No

method	Name of the method of the EJB or Java class that implements the operation if you specify with the `component` attribute that the operation is implemented with a stateless session EJB or Java class.
	You can specify all the methods with the asterisk (*) character.
	If your EJB or Java class does *not* overload the method, you need only specify the name of the method, such as:
	`method="sell"`
	If, however, the EJB or Java class overloads the method, then specify the full signature, such as:
	`method="sell(int)"`
		String	No

name	Name of the operation that will be used in the generated WSDL. If you do not specify this attribute, the name of the operation defaults to either the name of the method or the name of the SOAP message handler chain.	String	No

Table A-10 Attributes of the <operation> Element

Attribute	Description	Datatype	Required?
out-security-spec	Specifies the name of the security specification that describes the message-level security of WebLogic Server's SOAP response after a client application has invoked the operation. The security specification describes what part of the SOAP response should be encrypted or digitally signed. If you do not specify this attribute, the default security specification, if it exists, is applied to the SOAP response. If there is no default security specification, then no message-level security is applied to the SOAP response. The value of this attribute corresponds to the `Id` attribute of the appropriate `<spec:SecuritySpec>` element.	String	No.
portTypeName	Port type in the WSDL file to which this operation belongs. You can include this operation in multiple port types by specifying a comma-separated list of port types. When the WSDL for this Web Service is generated, a separate `<portType>` element is created for each specified port type. The default value is the value of the `portType` attribute of the `<web-service>` element.	String	No

operations

The `<operations>` element groups together the explicitly declared operations of this Web Service.

This element does not have any attributes.

outbound-expiry

Specifies, in milliseconds, the expiration period that WebLogic Server adds to the timestamp header of the SOAP response.

To specify no expiration, set this element to `-1`.

The default value of this element is `-1`.

param

The <param> element specifies a single parameter of an operation.

You must list the parameters in the same order in which they are defined in the method that implements the operation. The number of <param> elements must match the number of parameters of the method.

Table A-11 Attributes of the <param> Element

Attribute	Description	Datatype	Required?
class-name	Java class name of the Java representation of the data type of the parameter. If you do not specify this attribute, WebLogic Server introspects the back-end component that implements the operation for the Java class of the parameter. You are required to specify this attribute only if you want the mapping between the XML and Java representations of the parameter to be different than the default. For example, `xsd:int` maps to the Java primitive `int` type by default, so use this attribute to map it to `java.lang.Integer` instead.	NMTOKEN	Maybe. See the description of the attribute.
location	Part of the request SOAP message (either the header, the body, or the attachment) that contains the value of the input parameter. Valid values for this attribute are `Body`, `Header`, or `attachment`. The default value is `Body`. If you specify `Body`, the value of the parameter is extracted from the SOAP Body, according to regular SOAP rules for RPC operation invocation. If you specify `Header`, the value is extracted from a SOAP Header element whose name is the value of the `type` attribute. If you specify `attachment`, the value of the parameter is extracted from the SOAP Attachment rather than the SOAP envelope. As specified by the JAX-RPC specification, only the following Java data types can be extracted from the SOAP Attachment: • `java.awt.Image` • `java.lang.String` • `javax.mail.internet.MimeMultiport` • `javax.xml.transform.Source` • `javax.activation.DataHandler`	String	No.

Table A-11 Attributes of the <param> Element

Attribute	Description	Datatype	Required?
name	Name of the input parameter that will be used in the generated WSDL. The default value is the name of the parameter in the method's signature.	String	No.
style	Style of the input parameter, either a standard input parameter, an out parameter used as a return value, or an in-out parameter for both inputting and outputting values. Valid values for this attribute are `in`, `out`, and `in-out`. If you specify a parameter as `out` or `in-out`, the Java class of the parameter in the back-end component's method must implement the `javax.xml.rpc.holders.Holder` interface.	String	Yes.
type	XML Schema data type of the parameter.	NMTOKEN	Yes.

params

The `<params>` element groups together the explicitly declared parameters and return values of an operation.

You do not have to explicitly list the parameters or return values of an operation. If an `<operation>` element does not have a `<params>` child element, WebLogic Server introspects the back-end component that implements the operation to determine its parameters and return values. When generating the WSDL file of the Web Service, WebLogic Server uses the names of the corresponding method's parameters and return value.

You explicitly list an operation's parameters and return values when you want:

- The name of the parameters and return values in the generated WSDL to be different from those of the method that implements the operation.
- To map a parameter to a name in the SOAP header request or response.
- To use out or in-out parameters.

Use the `<param>` child element to specify the parameters of the operation.

Use the <return-param> child element to specify the return value of the operation.

The <params> element does not have any attributes.

password

Depending on the parent element, the <password> element specifies:

- The password used in the username token in the SOAP response message. (Parent element is <user>.)
- The password of the key pair and certificate, stored in WebLogic Server's keystore, used to encrypt part of the SOAP message. (Parent element is <encryptionKey>.)
- The password of the key pair and certificate, stored in WebLogic Server's keystore, used to digitally sign part of the SOAP message. (Parent element is <signatureKey>.)

This element does not have any attributes.

reliable-delivery

The <reliable-delivery> element specifies that the operation can be invoked asynchronously using reliable SOAP messaging. This means that the application that invokes the Web Service has a guaranteed that the SOAP message was delivered to the Web Service operation, or it receives an explicit exception saying that the delivery did not happen.

You can specify only one <reliable-delivery> element for a given operation.

Table A-12 Attributes of the <reliable-delivery> Element

Attribute	Description	Datatype	Required?
duplicate-elimination	Specifies whether the WebLogic Web Service should ignore duplicate invokes from the same client application. If this attribute is set to `True`, the Web Service persists the message IDs from client applications that invoke the Web Service so that it can eliminate any duplicate invokes. If this values is set to `False`, the Web Service does not keep track of duplicate invokes, which means that if a client retries an invoke, both invokes could return values. Valid values for this attribute are `True` and `False`. The default value is `True`.	Boolean	No.
persist-duration	The default minimum number of seconds that the Web Service should persist the history of a reliable SOAP message (received from the sender that invoked the Web Service) in its storage. The Web Service, after recovering from a WebLogic Server crash, does not dispatch persisted messages that have expired. The value of this attribute, if you set it, should be greater than the product of the retry interval and the retry count of the sender. The default value of this attribute is 60,000.	Integer	No.

require-signature-timestamp

Specifies whether WebLogic Server requires that the SOAP request from a client application that invokes a WebLogic Server include a timestamp. If this element is set to `true`, and a SOAP request does not contain a timestamp, WebLogic Server rejects the request.

Valid values for this element are `true` and `false`. The default value is `true`.

This element does not have any attributes. This element is a child element of `<timestamp>`.

return-param

The `<return-param>` element specifies the return value of the Web Service operation.

You can specify only one `<return-param>` element for a given operation.

Table A-13 Attributes of the <return-param> Element

Attribute	Description	Datatype	Required?
class-name	Java class name of the Java representation of the data type of the return parameter. If you do not specify this attribute, WebLogic Server introspects the back-end component that implements the operation for the Java class of the return parameter. You are required to specify this attribute if: • The back-end component that implements the operation is `<jms-receive-queue>`. • The mapping between the XML and Java representations of the return parameter is ambiguous, such as mapping `xsd:int` to either the `int` Java primitive type or `java.lang.Integer`.	NMTOKEN	Maybe. See the description of the attribute.
location	Part of the response SOAP message (either the header, the body, or the attachment) that contains the value of the return parameter. Valid values for this attribute are `Body`, `Header`, or `attachment`. The default value is `Body`. If you specify `Body`, the value of the return parameter is added to the SOAP Body. If you specify `Header`, the value is added as a SOAP Header element whose name is the value of the `type` attribute. If you specify `attachment`, the value of the parameter is added to the SOAP Attachment rather than the SOAP envelope. As specified by the JAX-RPC specification, only the following Java data types can be added to the SOAP Attachment: • `java.awt.Image` • `java.lang.String` • `javax.mail.internet.MimeMultiport` • `javax.xml.transform.Source` • `javax.activation.DataHandler`	String	No.

Table A-13 Attributes of the <return-param> Element

Attribute	Description	Datatype	Required?
name	Name of the return parameter that will be used in the generated WSDL file. If you do not specify this attribute, the return parameter is called `result`.	String	No.
type	XML Schema data type of the return parameter.	NMTOKEN	Yes.

security

Element that contains all the security configuration information about a particular Web Service. This information includes:

- The username and password used in the SOAP response username token (`<user>` child element).
- The name of the key pairs in WebLogic Server's keystore used for data encryption and digital signatures (`<encryptionKey>` and `<signatureKey>` child elements).
- What parts of the SOAP message should be encrypted and digitally signed (`<spec:SecuritySpec>` child element).

Table A-14 Attributes of the <security> Element

Attribute	Description	Datatype	Required?
Name	The name of this security element.	String	Yes.

signatureKey

Specifies the name and password of a key pair and certificate used when digitally signing elements of the SOAP message. Specify the name using the `<name>` subelement; specify the password with the `<password>` subelement.

Note: Create the key pair and certificate in the WebLogic Server keystore with the Administration Console. For details, see Storing Private Keys, Digital Certificates, and Trusted CAs.

This element does not have any attributes.

spec:BinarySecurityTokenSpec

Specifies the (binary) non-XML-based security tokens included in the SOAP messages.

Note: You must include the following namespace declaration with this element:

```
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
```

and the value of each attribute of this element should be qualified with the `wsse` namespace.

Table A-15 Attributes of the <spec:BinarySecurityTokenSpec> Element

Attribute	Description	Datatype	Required?
EncodingType	Specifies the encoding format of the binary data. Only one valid value: `wsse:Base64Binary`	String	Yes
ValueType	Specifies the value type and space of the encoded binary data. Only one valid value: `wsse:X509v3` (for X.509 certificates)	String	Yes

spec:ElementIdentifier

Identifies a particular element in the SOAP message (either the header or the body) that you want to digitally sign or encrypt. You uniquely identify an element in the SOAP message by its local name and its namespace.

Specify this element as the child of either `<spec:SignatureSpec>` or `<spec:EncryptionSpec>`.

Table A-16 Attributes of the <spec:ElementIdentifier> Element

Attribute	Description	Datatype	Required?
LocalPart	The local name of the element. Do not specify the namespace with this attribute.	String	Yes.
Namespace	The namespace in which the element is defined.	String	Yes.
Restriction	Specifies whether to restrict the identification of the element to the SOAP header or body. Valid values are `header` or `body`. If this attribute is not specified, the entire SOAP message is searched when identifying the element. **Note:** If you specify a value for this optional attribute, only the top-level elements in the relevant SOAP message part (header or body) are searched. If you do not specify this attribute, then all elements, no matter how deeply nested, are searched.	String	No.

spec:EncryptionSpec

Specifies the elements in the SOAP message that are encrypted and how they are encrypted.

You can specify that the entire SOAP body be encrypted by setting the attribute `EncryptBody="True"`. You can also use the `<spec:ElementIdentifier>` child element to specify particular elements of the SOAP message that are to be encrypted.

Warning: Do not specify both `EncryptBody="True"` and one or more elements with the `<spec:ElementIdentifier>` child element, but rather, use just one way to specify the elements of the SOAP message that should be encrypted.

Use the `EncryptionMethod` attribute to specify how to encrypt the SOAP message elements.

Table A-17 Attributes of the <spec:EncryptionSpec> Element

Attribute	Description	Data type	Required?
EncryptBody	Specifies whether to encrypt the entire SOAP body. **Note:** Do not specify both `EncryptBody="True"` and one or more elements with the `<spec:ElementIdentifier>` child element, but rather, use just one way to specify the elements of the SOAP message that should be encrypted. Valid values are `True` and `False`.	String	Yes.

EncryptionMethod	Specifies the algorithm used to encrypt the specified elements of the SOAP message.
	Valid values are:
	`http://www.w3.org/2001/04/xmlenc#tripledes-cbc`
	`http://www.w3.org/2001/04/xmlenc#kw-tripledes`
	Default value is `http://www.w3.org/2001/04/xmlenc#tripledes-cbc`.
		String	No.

KeyWrappingMethod	Specifies the algorithm used to encrypt the message encryption key. Valid values are: `http://www.w3.org/2001/04/xmlenc#rsa-1_5` (to specify the REQUIRED RSA-v1.5 algorithm) `http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p` (to specify the REQUIRED RSA-OAEP algorithm) When using this attribute, set the value to the URI, such as: `KeyWrappingMethod="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p"` Default value is `http://www.w3.org/2001/04/xmlenc#rsa-1_5`.	String	No.

spec:SecuritySpec

Specifies the set of security-related information associated with this Web Service.

The information in this element can include:

- A security token that specifies the username and password of the client invoking the Web Service. (`<spec:UsernameTokenSpec>` child element)
- A binary security token that specify non-XML-based security tokens, such as X.509 certificates. (`<spec:BinarySecurityTokenSpec>` child element)
- A specification for the parts of the SOAP message that are digitally signed. (`<spec:SignatureSpec>` child element)
- A specification of the parts of the SOAP message that are encrypted. (`<spec:EncryptionSpec>` child element.)

The information in the `<spec:SecuritySpec>` element appears in the generated WSDL of the Web Service so that client applications that invoke the Web Service know how to create the SOAP request to comply with all the security specifications.

WebLogic Server also uses the information in this element to verify that a SOAP request to invoke a particular Web Service contains all the necessary security information in the header. For example, if the `<spec:SecuritySpec>` element requires that a portion of the SOAP message be digitally signed, then WebLogic Server knows to check for this when it receives the SOAP request. WebLogic Server then uses the same information to create the security information in the SOAP response message.

You can create many security specifications for a single Web Service, and specify that different operations use different security specifications. For example, you can configure one operation so that the SOAP messages (both request and response) are only digitally signed, and configure a different operation such that only the SOAP request is both digitally signed and encrypted. You do this by associating operations with different security specifications.

Note: You must include the following namespace declaration with this element:

```
xmlns:spec="http://www.openuri.org/2002/11/wsse/spec"
```

and all child elements of the `<spec:SecuritySpec>` element must be qualified with the `spec` namespace.

Table A-18 Attributes of the <spec:SecuritySpec> Element

Attribute	Description	Datatype	Required?
Id	Name used to identity this security specification. This id can be later associated with an operation. If you do not specify this attribute, this security specification becomes the default for the Web Service. This means that operations that do not explicitly associate with a security specification use this one by default.	String	No.
Namespace	The namespace in which this security specification is defined.	String	Yes.

spec:SignatureSpec

Specifies the elements in the SOAP message that are digitally signed and how they are signed.

Digital signatures are a way to determine whether a message was altered in transit and to verify that a message was really sent by the possessor of a particular security token.

You can specify that the entire SOAP body be digitally signed by setting the attribute `SignBody="True"`. Use the `<spec:ElementIdentifier>` child element to specify additional particular elements of the SOAP message that are to be signed.

Use the `CanonicalizationMethod` and `SignatureMethod` attributes to specify how to digitally sign the SOAP message elements.

Table A-19 Attributes of the <spec:SignatureSpec> Element

Attribute	Description	Data type	Required?
CanonicalizationMethod	Specifies the algorithm used to canonicalize the SOAP message elements being signed. Only one valid value: `http://www.w3.org/2001/10/xml-exc-cl4n#`	String	Yes.

SignatureMethod	Specifies the cryptographic algorithm used to compute the signature.
	Note: Be sure that you specify an algorithm that is compatible with the certificates you are using in your enterprise.
	Valid values are:
	`http://www.w3.org/2000/09/xmldsig#rsa-sha1`
	`http://www.w3.org/2000/09/xmldsig#dsa-sha1`
		String	Yes.

SignBody	Specifies whether to digitally sign the entire SOAP body, in addition to the any specific elements identified with the optional `<spec:ElementIdentifier>` child elements. Valid values are `True` and `False`.	String	Yes.

spec:UsernameTokenSpec

Specifies that the SOAP response after an invoke of this Web Service must include a username token.

WebLogic Server uses the information in the `<user>` child element of the `<security>` element when creating the security information in a SOAP response message.

Note: You must include the following namespace declaration with this element:

```
xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/07/secext"
```

and the value of each attribute of this element should be qualified with the `wsse` namespace.

Table A-20 Attributes of the <spec:UsernameTokenSpec> Element

Attribute	Description	Datatype	Required?
PasswordType	Specifies how to include the password in the SOAP message. Only valid value is `wsse:PasswordText` (actual password for the username.)	String	Yes.

stateless-ejb

Describes the stateless session EJB component that implements one or more operations of a Web Service.

Table A-21 Attributes of the <stateless-ejb> Element

Attribute	Description	Datatype	Required?
name	Name of the stateless EJB component. **Note:** The name is internal to the `web-services.xml` file; it does not refer to the name of the EJB in the `ejb-jar.xml` file.	String	Yes.

timestamp

The `<timestamp>` element groups together the elements used to configure timestamp behavior of WebLogic Server. WebLogic Server adds or requires timestamps only in SOAP messages that are encrypted or digitally signed.

If the `web-services.xml` deployment descriptor does not include a `<timestamp>` child element of the `<security>` element, and the Web Service has been configured for message-level security (encryption and digital signatures), WebLogic Server:

- Requires that SOAP requests include a timestamp and rejects any that do not.
- Assumes that its clock and the client application's clock are *not* synchronized. This means that if the SOAP request from a client application includes a timestamp with an expiration,

WebLogic Server rejects the message because it cannot ensure that the message has not already expired.

Adds a timestamp to the SOAP response. The timestamp contains only the creation date of the SOAP response.

This element has no attributes. This element has the following child elements:

- clock-precision
- clocks-synchronized
- enforce-precision
- generate-signature-timestamp
- inbound-expiry
- outbound-expiry
- require-signature-timestamp

type-mapping

The `<type-mapping>` element contains the list of mappings between the XML data types defined in the `<types>` element and their Java representations.

For each data type in the `<types>` element, there is a corresponding `<type-mapping-entry>` element that lists the Java class that implements the data type, how to serialize and deserialize the data, and so on.

This element has no attributes.

type-mapping-entry

Describes the mapping between a single XML data type in the `<types>` element and its Java representation.

Table A-22 Attributes of the <type-mapping-entry> Element

Attribute	Description	Datatype	Required?
class-name	Fully qualified name of the Java class that maps to its corresponding XML data type.	String	Yes.
deserializer	Fully qualified name of the Java class that converts the data from XML to Java.	String	Only required if the data type is *not* one of the built-in data dates supported by the WebLogic Web Services runtime, listed in "Supported Built-In Data Types" on page 5-15.
element	Name of the XML data type that maps to the Java data type. Specify only if the XML Schema definition of the data type uses the `<element>` element.	NMTOKEN	One, but not both, of either `element` or `type` is required.
serializer	Fully qualified name of the Java class that converts the data from Java to XML.	String	Only required if the data type is *not* one of the built-in data dates supported by the WebLogic Web Services runtime, listed in "Supported Built-In Data Types" on page 5-15.
type	Name of the XML data type that maps to the Java data type. Specify only if the XML Schema definition of the data type uses the `<type>` element.	NMTOKEN	One, but not both, of either `element` or `type` is required.

types

Describes, using XML Schema notation, the non-built-in data types used as parameters or return types of the Web Service operations.

For details on using XML Schema to describe the XML representation of a non-built-in data type, see http://www.w3.org/TR/xmlschema-0/.

The following example shows an XML Schema declaration of a data type called `TradeResult` that contains two elements: `stockSymbol`, a string data type, and `numberTraded`, an integer.

```
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
              xmlns:stns="java:examples.webservices"
              attributeFormDefault="qualified"
              elementFormDefault="qualified"
              targetNamespace="java:examples.webservices">
      <xsd:complexType name="TradeResult">
          <xsd:sequence>
              <xsd:element maxOccurs="1"
                           name="stockSymbol"
                           type="xsd:string" minOccurs="1">
              </xsd:element>
              <xsd:element maxOccurs="1"
                           name="numberTraded"
                           type="xsd:int"
                           minOccurs="1">
              </xsd:element>
          </xsd:sequence>
      </xsd:complexType>
  </xsd:schema>
</types>
```

user

Specifies the username and password to be used in the SOAP response message.

This element has two child elements:

- `<name>`
- `<password>`

This element has no attributes.

web-service

Defines a single Web Service.

The Web Service is defined by the following:

- Back-end components that implement an operation, such as a stateless session EJB, a Java class, or a JMS consumer or producer.
- An optional set of data type declarations for non-built-in data types used as parameters or return values to the Web Service operations.
- An optional set of XML to Java data type mappings that specify the serialization class and Java classes for the non-built-in data types.
- A declaration of the operations supported by the Web Service.

Table A-23 Attributes of the <web-service> Element

Attribute	Description	Datatype	Required?
charset	Specifies the character set that the Web Service uses in its response to an invocation. Examples of character sets include `US-ASCII`, `UTF-8`, and `Shift_JIS`. The default value is `US-ASCII`. **Note:** Although `US-ASCII` is the default value for this attribute, this does not mean that the `US-ASCII` character set will always be used in the response of a WebLogic Web Service invocation if you have not explicitly set the attribute in the `web-service.xml` file of the Web Service. See "Order of Precedence of Character Set Configuration Used By WebLogic Server" for more details.	String	No.
exposeHomePage	Specifies whether to publicly expose the Home Page of the Web Service. Valid values for this attribute are `True` and `False`. The default value is `True`. This means that by default the Home Page is publicly accessible.	Boolean	No.

Table A-23 Attributes of the <web-service> Element

Attribute	Description	Datatype	Required?
exposeWSDL	Specifies whether to publicly expose the automatically generated WSDL of the Web Service. Valid values for this attribute are `True` and `False`. The default value is `True`. This means that by default the WSDL is publicly accessible.	Boolean	No.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request. **Note:** Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it. Valid values are `True` and `False`. Default value is `False`.	Boolean.	No.
jmsUri	Specifies that client applications can use the JMS transport to invoke the Web Service, in addition to the default HTTP/S transport. The form of this attribute is: *`connection_factory_name`*/*`queue_name`* where *`connection_factory_name`* is the JNDI name of the JMS connection factory and *`queue_name`* is the JNDI name of the JMS queue that you have configured for the JMS transport. For example: `jmsURI="JMSTransFactory/JMSTransQueue"` If this attribute is set, the generated WSDL of the Web Service contains an additional port that uses a JMS binding. The `clientgen` Ant task, when generating the stubs used to invoke this Web Service, generates a `get`*`Service`*`PortJMS()` method, in addition to the default `get`*`Service`*`Port()` method, used for JMS and HTTP/S respectively.	String.	No.

Table A-23 Attributes of the <web-service> Element

Attribute	Description	Datatype	Required?
name	Name of the Web Service.	String	Yes.
portName	Name of the `<port>` child element of the `<service>` element of the dynamically generated WSDL of this Web Service. The default value is the `name` attribute of this element with `Port` appended. For example, if the name of this Web Service is `TraderService`, the port name will be `TraderServicePort`.	String	No
portTypeName	Name of the default `<portType>` element in the dynamically generated WSDL of this Web Service. The default value is the `name` attribute of this element with `Port` appended. For example, if the name of this Web Service is `TraderService`, the portType name will be `TraderServicePort`.	String	No.
protocol	Protocol over which the service is invoked. Valid values are `http` or `https`. Default is `http`.	String	No.

Table A-23 Attributes of the <web-service> Element

Attribute	Description	Datatype	Required?
style	Specifies whether the Web Service has RPC-oriented or document-oriented operations. RPC-oriented WebLogic Web Service operations use SOAP encoding. Document-oriented WebLogic Web Service operations use literal encoding. The following two values specify document-oriented Web Service operations: `document` and `documentwrapped`. If you specify `document` for this attribute, the resulting Web Service operations take only one parameter. This means that the methods that implement the operations must also have only *one* parameter. If you specify `documentwrapped`, the resulting Web Service operations can take any number of parameters, although the parameter values will be wrapped into one complex data type in the SOAP messages. If two or more methods of your stateless session EJB or Java class that implement the Web Service have the same number and data type of parameters, and you want the operations to be document-oriented, you must specify `documentwrapped` for this attribute rather than `document`. Valid values are `rpc`, `documentwrapped`, and `document`. Default value is `rpc`. **Note:** Because the `style` attribute applies to an entire Web Service, all operations specified in a single `<web-service>` element must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web Service.	String	No.
targetNamespace	Namespace of this Web Service.	String	Yes.

Table A-23 Attributes of the <web-service> Element

Attribute	Description	Datatype	Required?
uri	URI of the Web Service, used subsequently in the URL that invokes the Web Service. **Note:** Be sure to specify the leading "/", such as `/TraderService`.	String	Yes.
useSOAP12	Specifies whether to use SOAP 1.2 as the message format protocol. By default, WebLogic Web Services use SOAP 1.1. If you specify `useSOAP12="True"`, the generated WSDL of the deployed WebLogic Web Service includes *two* ports: the standard port that specifies a binding for SOAP 1.1 as the message format protocol, and a second port that uses SOAP 1.2. Client applications, when invoking the Web Service, can use the second port if they want to use SOAP 1.2 as their message format protocol. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.

web-services

The root element of the `web-services.xml` deployment descriptor.

This element does not have any attributes.

APPENDIX B

Web Service Ant Tasks and Command-Line Utilities

The following sections describe WebLogic Web Service Ant tasks and the command-line utilities based on these Ant tasks:

- "Overview of WebLogic Web Services Ant Tasks and Command-Line Utilities" on page B-1
- "autotype" on page B-7
- "clientgen" on page B-14
- "servicegen" on page B-25
- "source2wsdd" on page B-41
- "wsdl2Service" on page B-46
- "wsdlgen" on page B-50
- "wspackage" on page B-52

Overview of WebLogic Web Services Ant Tasks and Command-Line Utilities

Ant is a Java-based build tool, similar to the `make` command but much more powerful. Ant uses XML-based configuration files (called `build.xml` by default) to execute tasks written in Java.

BEA provides a number of Ant tasks that help you generate important parts of a Web Service (such as the serialization class, a client JAR file, and the `web-services.xml` file) and to package all the pieces of a WebLogic Web Service into a deployable EAR file.

The Apache Web site provides other useful Ant tasks for packaging EAR, WAR, and EJB JAR files. For more information, see http://jakarta.apache.org/ant/manual/.

You can also run some of the Ant tasks as a command-line utility, using flags rather than attributes to specify how the utility works. The description of the flags is exactly the same as the description of its corresponding attribute.

Warning: Not all the attributes of the Ant tasks are available as flags to the equivalent command-line utility. See the sections that describe each Ant task for a list of the supported flags when using the command-line equivalent.

For further examples and explanations of using these Ant tasks, see Chapter 6, "Assembling WebLogic Web Services Using Ant Tasks."

List of Web Services Ant Tasks and Command-Line Utilities

The following table provides an overview of the Web Service Ant tasks provided by BEA and the name of the corresponding command-line utility.

Table B-1 WebLogic Web Services Ant Tasks

Ant Task	Corresponding Command-Line Utility	Description
autotype	weblogic.webservice.autotype	Generates the serialization class, Java representation, XML Schema representation, and data type mapping information for non-built-in data types used as parameters or return values to a WebLogic Web Service.
clientgen	weblogic.webservice.clientgen	Generates a client JAR file that contains a thin Java client used to invoke a Web Service.

Table B-1 WebLogic Web Services Ant Tasks

Ant Task	Corresponding Command-Line Utility	Description
servicegen	Not available.	Main Ant task that performs all the steps needed to assemble a Web Service. These steps include: • Creating the Web Service deployment descriptor (`web-services.xml`). • Introspecting EJBs and Java classes and generating any needed non-built-in data type supporting components. • Generating the client JAR file. • Packaging all the pieces into a deployable EAR file.
source2wsdd	Not available	Generates a `web-services.xml` deployment descriptor file from the Java source file for a Java class-implemented WebLogic Web Service.
wsdl2Service	Not available.	Generates the components of a WebLogic Web Service from a WSDL file. The components include the `web-services.xml` deployment descriptor file and a Java source file that you can use as a starting point to implement the Web Service.
wsdlgen	weblogic.webservice.wsdlgen	Generates a WSDL file from the EAR and WAR files that make up the Web Service.
wspackage	Not available.	Packages the components of a WebLogic Web Service into a deployable EAR file.

Using the Web Services Ant Tasks

To use the Ant tasks, follow these steps:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is *`BEA_HOME`*`\user_projects\domains\domainName`, where *`BEA_HOME`* is the top-level installation directory of the BEA products and *`domainName`* is the name of your domain.

   On UNIX, execute the `setEnv.sh` command, located in your domain directory. The default location of WebLogic Server domains is

BEA_HOME/user_projects/domains/domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Create a file called `build.xml` that contains a call to the Web Services Ant tasks.

   The following example shows a simple `build.xml` file (with details of the Web Services Ant tasks `servicegen` and `clientgen` omitted for clarity):

```
<project name="buildWebservice" default="build-ear">
  <target name="build-ear">
    <servicegen attributes go here...>
      ...
    </servicegen>
  </target>
  <target name="build-client" depends="build-ear">
    <clientgen attributes go here .../>
  </target>
  <target name="clean">
    <delete>
      <fileset dir="."
               includes="example.ear,client.jar" />
    </delete>
  </target>
</project>
```

   Later sections provide examples of specifying the Ant task in the `build.xml` file.

3. Execute the Ant task or tasks specified in the `build.xml` file by typing `ant` in the same directory as the `build.xml` file:

```
prompt> ant
```

Differences in Operating System Case Sensitivity When Manipulating WSDL and XML Schema Files

Many of the WebLogic Web Service Ant tasks have attributes that you can use to specify an operating system file, such as a WSDL or an XML Schema file. For example, you can use the `wsdl` attribute of the `clientgen` Ant task to create the Web Services-specific client JAR file from an existing WSDL file that describes a Web Service.

The Ant tasks process these files in a case-sensitive way. This means that if, for example, the XML Schema file specifies two complex types whose names differ only in their capilatization (for example, `MyReturnType` and `MYRETURNTYPE`), the `clientgen` Ant task correctly generates two separate sets of Java source files for the Java represenation of the complex data type: `MyReturnType.java` and `MYRETURNTYPE.java`.

However, compiling these source files into their respective class files might cause a problem if you are running the Ant task on Microsoft Windows, because Windows is a case *insensitive* operating system. This means that Windows considers the files `MyReturnType.java` and `MYRETURNTYPE.java` to have the same name. So when you compile the files on Windows, the second class file overwrites the first, and you end up with only one class file. The Ant tasks, however, expect that *two* classes were compiled, thus resulting in an error similar to the following:

```
c:\src\com\bea\order\MyReturnType.java:14:
class MYRETURNTYPE is public, should be declared in a file named
MYRETURNTYPE.java
public class MYRETURNTYPE
       ^
```

To work around this problem rewrite the XML Schema so that this type of naming conflict does not occur, or if that is not possible, run the Ant task on a case sensitive operating system, such as Unix.

Setting the Classpath for the WebLogic Ant Tasks

Each WebLogic Ant task accepts a `classpath` attribute or element so that you can add new directories or JAR files to your current CLASSPATH environment variable.

The following example shows how to use the `classpath` attribute of the `servicegen` Ant task to add to the CLASSPATH variable:

```
<servicegen destEar="myEJB.ear"
            classpath="${java.class.path};my_fab_directory"
   ...
</servicegen>
```

The following example shows how to add to the CLASSPATH by using the `<classpath>` element:

```
<servicegen ...>
   <classpath>
       <pathelement path="${java.class.path}" />
       <pathelement path="my_fab_directory" />
   </classpath>
...
</servicegen>
```

The following example shows how you can build your CLASSPATH variable outside of the WebLogic Web Service Ant task declarations, then specify the variable from within the task using the `<classpath>` element:

```
<path id="myid">
   <pathelement path="${java.class.path}"/>
   <pathelement path="${additional.path1}"/>
   <pathelement path="${additional.path2}"/>
</path>

<servicegen ....>
   <classpath refid="myid" />
...
</servicegen>
```

Warning: The WebLogic Web Services Ant tasks support the standard Ant property `build.sysclasspath`. The default value for this property is `ignore`. This means that if you specifically set the CLASSPATH in the `build.xml` file as described in this section, the Ant task you want to run ignores the system CLASSPATH (or the CLASSPATH in effect when Ant is run) and uses only the one that you specifically set. It is up to you to include in your CLASSPATH setting all the classes that the Ant task needs to successfully run. To change this default behavior, set the `build.sysclasspath` property to `last` to concatenate the system CLASSPATH to the end of the one you specified, or `first` to concatenate your specified CLASSPATH to the end of the system one.

For more information on the `build.sysclasspath` property, see the Ant documentation.

Note: The Java Ant utility included in WebLogic Server uses the `ant` (UNIX) or `ant.bat` (Windows) configuration files in the *`WL_HOME`*`\server\bin` directory to set various Ant-specific variables, where *`WL_HOME`* is the top-level directory of your WebLogic Platform installation If you need to update these Ant variables, make the relevant changes to the appropriate file for your operating system.

Using the Web Services Command-Line Utilities

To use the command-line utility equivalents of the Ant tasks, follow these steps:

1. Set your environment.

   On Windows NT, execute the `setEnv.cmd` command, located in your domain directory. The default location of WebLogic Server domains is

BEA_HOME\user_projects\domains\domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

On UNIX, execute the setEnv.sh command, located in your domain directory. The default location of WebLogic Server domains is *BEA_HOME*/user_projects/domains/domainName, where *BEA_HOME* is the top-level installation directory of the BEA products and *domainName* is the name of your domain.

2. Open a command shell window.
3. Execute the utility using the java command, as shown in the following example:

```
prompt> java weblogic.webservice.clientgen \
               -ear myapps/myapp.ear \
               -serviceName myService \
               -packageName myservice.client \
                -clientJar myapps/myService_client.jar
```

Run the command with no arguments to get a usage message.

autotype

The autotype Ant task generates the following components for non-built-in data types that used as parameters or return values of your Web Service operation:

- Serialization class that converts between the XML and Java representation of the data.
- Given an XML Schema or WSDL file, a Java class to contain the Java representation of the data type.
- Given a Java class that represents the non-built-in data type, an XML Schema representation of the data type.
- Data type mapping information to be included in the web-services.xml deployment descriptor file.

For the list of non-built-in data types for which autotype can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.

You can specify one of the following types of input to the autotype Ant task:

- A Java class file that represents your non-built-in data types by specifying the javaTypes attribute. The autotype Ant task generates the corresponding XML Schemas, the serialization classes, and the data type mapping information for the web-services.xml file.

- A Java class file of the component that implements your Web Service by specifying the `javaComponents` attribute. If your Web Service is implemented with a Java class, then this attribute points to the Java class. If your Web Service is implemented with a stateless session EJB, then this attribute points to the remote interface of the EJB. The `autotype` Ant task looks for non-built-in data types used as parameters or return values in the component, then generates the corresponding XML Schemas, the serialization classes, and the data type mapping information for the `web-services.xml` file. Be sure to include the Java class for your non-built-in data type in your CLASSPATH.
- An XML Schema file that represents your non-built-in data type by specifying the `schemaFile` attribute. The `autotype` Ant task generates the corresponding Java representations, the serialization classes, and the data type mapping information for the `web-services.xml` file.
- A URL to a WSDL file that contains a description of your non-built-in data type by specifying the `wsdlURI` attribute. The `autotype` Ant task generates the corresponding Java representations, the serialization classes, and the data type mapping information for the `web-services.xml` file.

Use the `destDir` attribute to specify the name of a directory that contains the generated components. The generated XML Schema and data type mapping information are generated in a file called `types.xml`. You can use this file to manually update an existing `web-services.xml` file with non-built-in data type mapping information, or use it in conjunction with the `typeMappingFile` attribute of the `servicegen` or `clientgen` Ant tasks, or the `typesInfo` attribute of the `source2wsdd` Ant task.

Warning: The serialization class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

Note: The fully qualified name for the `autotype` Ant task is `weblogic.ant.taskdefs.webservices.javaschema.JavaSchema`.

Example

The following example shows how to create non-built-in data type components from a Java class:

```
<autotype  javatypes="mypackage.MyType"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
```

```
            destDir="output" />
```

The following example is similar to the preceding one, except it creates non-built-in data type components for an *array* of `mypackage.MyType` Java data types:

```
<autotype  javaTypes="[Lmypackage.MyType;"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="output" />
```

Note: The `[L`*`classname`*`;` syntax follows the Java class naming conventions as outlined in the `java.lang.Class.getName()` method documentation.

The following example shows how to use the `autotype` Ant task against a WSDL file; it also shows how to specify that the Ant task keep the generated Java files for the serialization class:

```
<autotype  wsdl="wsdls/myWSDL"
           targetNamespace="http://www.foobar.com/autotyper"
           packageName="a.package.name"
           destDir="output"
           keepGenerated="True" />
```

Attributes

The following table describes the attributes of the `autotype` Ant task.

Table B-2 Attributes of the autotype Ant task

Attribute	Description	Data Type	Required?
destDir	Full pathname of the directory that will contain the generated components. The generated XML Schema and data type mapping information are generated in a file called `types.xml`.	String	Yes.
encoding	Specifies whether the `autotype` Ant task uses SOAP or literal encoding when generating the XML Schema file that describes the XML representation of a Java data type. The encoding is particularly important when generating complex XML data types, such as arrays. SOAP arrays are structurally different from non-SOAP arrays, so it is important to always use the correct one for the correct situation. If you are creating a document-oriented Web Service, you *must* specify that `autotype` use literal encoding, or users might run into interoperability problems when they later invoke your Web Service. Use this attribute only when generating an XML Schema from an existing Java data type. Valid values for this attribute are `soap` and `literal`. The default value is `soap`.	String	No. Use this attribute only in conjunction with either the `javaTypes` or `javaComponents` attributes. An error is returned if you use the `encoding` attribute with the `schemaFile` attribute.
javaComponents	Comma-separated list of Java class names that implement the Web Service. For Java class-implemented Web Services, this attribute points to the Java class. For stateless session EJB-implemented Web Services, this attribute points to the remote interface of the EJB. The Java classes (of both the implementation of the component and the implementation of your non-built-in data type) must be compiled and in your CLASSPATH. For example: `javaComponents="my.class1,my.class2"` The `autotype` Ant task introspects the Java classes to automatically generate the components for all non-built-in data types it finds.	String	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.

Table B-2 Attributes of the autotype Ant task

Attribute	Description	Data Type	Required?
javaTypes	Comma-separated list of Java class names that represent your non-built-in data types. The Java classes must be compiled and in your CLASSPATH. For example: `javaTypes="my.class1,my.class2"` **Note:** Use the syntax `[Lclassname;` to specify an array of the Java data type. For an example, see "Example" on page B-8.	String	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.
keepGenerated	Specifies whether the `autotype` Ant task should keep (and thus include in the generated components) the Java source code of the serialization class for any non-built-in data types used as parameters or return values to the Web Service operations, or whether the `autotype` Ant task should include only the compiled class file. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.

Table B-2 Attributes of the autotype Ant task

Attribute	Description	Data Type	Required?
packageBase	Base package name of the generated Java classes for any non-built-in data types used as a return value or parameter in a Web Service. This means that each generated Java class will be part of the same package name, although the `autotype` Ant task generates its own specific name for each Java class which it appends to the specified package base name. If you do not specify this attribute, the `autotype` Ant task generates a base package name for you. **Note:** BEA recommends you not use this attribute, but rather, specify the full package name using the `packageName` attribute. The `packageBase` attribute is available for JAX-RPC compliance.	String	No. If you specify this attribute, you cannot also specify `packageName`.
packageName	Full package name of the generated Java classes for any non-built-in data types used as a return value or parameter in a Web Service. If you do not specify this attribute, the `autotype` Ant task generates a package name for you. **Note:** Although not required, BEA recommends you specify this attribute in most cases. Currently, the only situation in which you should *not* specify this attribute is if you use the `javaTypes` attribute to specify a list of Java data types whose class names are the same, but their package names are different. In this case, if you also specify the `packageName` attribute, the `autotype` Ant task generates a serialization class for only the last class.	String	No. If you specify this attribute, you cannot also specify `packageBase`.

Table B-2 Attributes of the autotype Ant task

Attribute	Description	Data Type	Required?
schemaFile	Name of a file that contains the XML Schema representation of your non-built-in data types.	String	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.
targetNamespace	Namespace URI of the generated XML Schema.	String	Yes.
typeMappingFile	File that contains data type mapping information for non-built-in data types for which have already generated needed components, as well as the XML Schema representation of your non-built-in data types. The format of the information is the same as the data type mapping information in the `<type-mapping>` and `<types>` elements of the `web-services.xml` file. The `autotype` Ant task does not generate non-built-in data type components for any data types listed in this file. It merges the information in this file with the generated information in its output `types.xml` file.	String	No.
wsdl	Full path name or URI of the WSDL that contains the XML Schema description of your non-built-in data type.	String	You must specify one, and only one, of the following attributes: `schemaFile`, `wsdl`, `javaTypes`, or `javaComponents`.

Equivalent Command-Line Utility

The equivalent command-line utility of the `autotype` Ant task is called `weblogic.webservice.autotype`. The description of the flags of the utility is the same as the description of the Ant task attributes, described in the preceding section.

The `weblogic.webservice.autotype` utility supports the following flags (see the equivalent attribute for a description of the flag):

- `-help (Prints the standard usage message)`
- -version (Prints version information)
- -verbose (Enables verbose output)
- -destDir *`dir`*
- -schemaFile *`pathname`*
- -wsdl *`uri`*
- -javaTypes *`classname`*
- -javaComponents *`classname`*
- -packageName *`name`*
- -packageBase *`name`*
- -encoding *`name`*
- -generatePublicFields *`true_or_false`*
- -typeMappingFile *`pathname`*
- -keepGenerated *`true_or_false`*

clientgen

The `clientgen` Ant task generates a Web Service-specific client JAR file that client applications can use to invoke both WebLogic and non-WebLogic Web Services. Typically, you use the `clientgen` Ant task to generate a client JAR file from an existing WSDL file; you can also use it with an EAR file that contains the implementation of a WebLogic Web Service.

The contents of the client JAR file includes:

- Client interface and stub files (conforming to the JAX-RPC specification) used to invoke a Web Service in static mode.
- Optional serialization class for converting non-built-in data between its XML and Java representation.
- Optional client-side copy of the Web Service WSDL file

You can use the `clientgen` Ant task to generate a client JAR file from the WSDL file of an existing Web Service (not necessarily running on WebLogic Server) or from an EAR file that contains a Weblogic Web Service implementation.

The WebLogic Server distribution includes a client runtime JAR file that contains the client side classes needed to support the WebLogic Web Services runtime component. For more information, see "Generating the Client JAR File by Running the clientgen Ant Task" on page 7-5.

Warning: The `clientgen` Ant task does not support solicit-response or notification WSDL operations. This means that if you attempt to create a client JAR file from a WSDL file that contains these types of operations, the Ant task ignores the operations.

Warning: The serialization class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

Note: The fully qualified name of the `clientgen` Ant task is `weblogic.ant.taskdefs.webservices.clientgen.ClientGenTask`.

Example

```
<clientgen wsdl="http://example.com/myapp/myservice.wsdl"
           packageName="myapp.myservice.client"
           clientJar="myapps/myService_client.jar"
/>
```

Attributes

The following table describes the attributes of the `clientgen` Ant task.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
autotype	Specifies whether the `clientgen` task should generate and include in the client JAR file the serialization class for any non-built-in data types used as parameters or return values to the Web Service operations. Valid values are `True` and `False`. Default value is `True`.	Boolean	No.
clientJar	Name of a JAR file or exploded directory into which the `clientgen` task puts the generated client interface classes, stub classes, optional serialization class, and so on. To create or update a JAR file, use a `.jar` suffix when specifying the JAR file, such as `myclientjar.jar`. If the attribute value does not have a `.jar` suffix, then the `clientgen` task assumes you are referring to a directory name. If you specify a JAR file or directory that does not exist, the `clientgen` task creates a new JAR file or directory.	String	Yes.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
ear	Name of an EAR file or exploded directory that contains the WebLogic Web Service implementation for which a client JAR file should be generated. **Note:** If the `saveWSDL` attribute of `clientgen` is set to `True` (the default value), the `clientgen` Ant task generates a WSDL file from the information in the EAR file, and stores it in the generated client JAR file. Because `clientgen` does not know the host name or port number of the WebLogic Server instance which will host the Web Service, `clientgen` uses the following endpoint address in the generated WSDL: `http://localhost:7001/`*`contextURI`*`/`*`serviceURI`* where *`contextURI`* and *`serviceURI`* are the same values as described in "WebLogic Web Services Home Page and WSDL URLs" on page 6-21. If this endpoint address is not correct, and your client application uses the WSDL file stored in the client JAR file, you must manually update the WSDL file with the correct endpoint address.	String	Either `wsdl` or `ear` must be specified.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?

generateAsyncMethods	Specifies that the `clientgen` Ant task should generate two special methods used to invoke each Web Service operation asynchronously, in addition to the standard methods. The special methods take the following form:
	
	`FutureResult start`*`Method`*` (`*`params`*`,`
	`AsyncInfo asyncInfo);`
	`result`` end`*`Method`*` (FutureResult`
	`futureResult);`
	
	where:
	• *`Method`* is the name of the standard method used to invoke the Web Service operation.
	• *`params`* is the list of parameters to the operation.
	• *`result`* is the result of the operation.
	• `FutureResult` is a WebLogic object used as a placeholder for the impending result.
	• `AsyncInfo` is a WebLogic object used to pass contextual information.
	
	Valid values for this attribute are `True` and `False`. The default value is `False.`
		Boolean	No.

generatePublicFields	Specifies whether the clientgen Ant task, when generating the Java representation of any non-built-in data types used by the Web Service, should use public fields for the JavaBean attributes rather than getter and setter methods. Valid values are `True` and `False`. Default values is `False`.	Boolean	No.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
j2me	Specifies whether the `clientgen` Ant task should create a J2ME/CDC-compliant client JAR file. **Note:** The generated client code is not JAX-RPC compliant. Valid values are `True` and `False`. Default value is `False`.	Boolean	No.
keepGenerated	Specifies whether the `clientgen` Ant task should keep (and thus include in the generated JAR file) the Java source code of the serialization class for any non-built-in data types used as parameters or return values to the Web Service operations, or whether the `clientgen` Ant task should include only the compiled class file. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
packageName	Package name into which the generated JAX-RPC client interfaces and stub files should be packaged.	String	Yes.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
saveWSDL	When set to `True`, specifies that the WSDL of the Web Service be saved in the generated client JAR file. This means that client applications do not need to download the WSDL every time they create a stub to the Web Service, possibly improving performance of the client because of reduced network usage. Valid values are `True` and `False`. Default value is `True`.	Boolean	No.
serviceName	Web Service name for which a corresponding client JAR file should be generated. If you specify the `wsdl` attribute, the Web Service name corresponds to the `<service>` elements in the WSDL file. If you specify the `ear` attribute, the Web Service name corresponds to the `<web-service>` element in the `web-services.xml` deployment descriptor file. If you do not specify the `serviceName` attribute, the `clientgen` task generates client classes for the first service name found in the WSDL or `web-services.xml` file.	String	No.
typeMappingFile	File that contains data type mapping information, used by the `clientgen` task when generating the JAX-RPC stubs. The format of the information is the same as the data type mapping information in the `<type-mapping>` element of the `web-services.xml` file. If you specified the `ear` attribute, the information in this file overrides the data type mapping information found in the `web-services.xml` file.	String	No.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
typePackageBase	Specifies the base package name of the generated Java class for any non-built-in data types used as a return value or parameter in a Web Service. This means that each generated Java class will be part of the same package name, although the `clientgen` Ant task generates its own specific name for each Java class which it appends to the specified package base name. If you specify this attribute, you cannot also specify `typePackageName`. If you do not specify this attribute *and* the XML Schema in the WSDL file defines a target namespace, then the `clientgen` Ant task generates a package name for you based on the target namespace. This means that if your XML Schema does *not* define a target namespace, then you must specify either the `typePackageName` (preferred) or `typePackageBase` attributes of the `clientgen` Ant task. **Note:** Rather than using this attribute, BEA recommends that you specify the full package name with the `typePackageName` attribute. The `typePackageBase` attribute is available for JAX-RPC compliance.	String	Required *only* if you specified the `wsdl` attribute and the XML Schema in the WSDL file does not define a target namespace.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
typePackageName	Specifies the full package name of the generated Java class for any non-built-in data types used as a return value or parameter in a Web Service. If you specify this attribute, you cannot also specify `typePackageBase`. If you do not specify this attribute *and* the XML Schema in the WSDL file defines a target namespace, then the `clientgen` Ant task generates a package name for you based on the target namespace. This means that if your XML Schema does *not* define a target namespace, then you must specify either the `typePackageName` (preferred) or `typePackageBase` attributes of the `clientgen` Ant task. **Note:** Although not required, BEA recommends you specify this attribute.	String	Required *only* if you specified the `wsdl` attribute and the XML Schema in the WSDL file does not define a target namespace..
useLowerCaseMethodNames	When set to true, specifies that the method names in the generated stubs have a lower-case first character. Otherwise, all method names will the same as the operation names in the WSDL file. Valid values are `True` and `False`. Default value is `True`.	Boolean	No.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
usePortNameAsMethodName	Specifies where the `clientgen` Ant task should get the names of the operations when generating a client from a WSDL file. If this value is set to true, then operations take the name specified by the `name` attribute of the `<port>` element in the WSDL file (where `<port>` is the child element of the `<service>` element). If `usePortNameAsMethodName` is set to false, then operations take the name specified by the `name` attribute of the `<portType>` element in the WSDL file (where `<portType>` is the child element of the `<definitions>` element). Valid values are `True` and `False`. Default value is `False`.	Boolean	No.
useServerTypes	Specifies where the `clientgen` task gets the implementation of any non-built-in Java data types used in a Web Service: either the task generates the Java code or the task gets it from the EAR file that contains the full implementation of the Web Service. Valid values are `True` (use the Java code in the EAR file) and `False`. Default value is `False`. For the list of non-built-in data types for which `clientgen` can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.	Boolean	No. Use only in combination with the `ear` attribute.

Table B-3 Attributes of the clientgen Ant Task

Attribute	Description	Data Type	Required?
warName	Name of the WAR file which contains the Web Service(s). The default value is `web-services.war`.	String	No. You can specify this attribute only in combination with the `ear` attribute.
wsdl	Full path name or URL of the WSDL that describes a Web Service (either WebLogic or non-WebLogic) for which a client JAR file should be generated. The generated stub factory classes in the client JAR file use the value of this attribute in the default constructor.	String	Either `wsdl` or `ear` must be specified.

Equivalent Command-Line Utility

The equivalent command-line utility of the `clientgen` Ant task is called `weblogic.webservice.clientgen`. The description of the flags of the utility is the same as the description of the Ant task attributes, described in the preceding section.

The `weblogic.webservice.clientgen` utility supports the following flags (see the equivalent attribute for a description of the flag):

- `-help (Prints the standard usage message)`
- -version (Prints version information)
- -verbose (Enables verbose output)
- `-wsdl` *`uri`*
- `-ear` *`pathname`*
- `-clientJar` *`pathname`*
- `-packageName` *`name`*

- `-warName` *`name`*
- `-serviceName` *`name`*
- `-typeMappingFile` *`pathname`*
- `-useServerTypes` *`true_or_false`*
- -typePackageName *`name`*
- -typePackageBase *`name`*
- -useLowerCaseMethodNames *`true_or_false`*
- -usePortNameAsMethodName *`true_or_false`*
- -generateAsyncMethods *`true_or_false`*
- -generatePublicFields *`true_or_false`*
- -saveWSDL *`true_or_false`*
- -autotype *`true_or_false`*
- -overwrite *`true_or_false`*
- -keepGenerated *`true_or_false`*

servicegen

The `servicegen` Ant task takes as input an EJB JAR file or list of Java classes, and creates all the needed Web Service components and packages them into a deployable EAR file.

In particular, the `servicegen` Ant task:

- Introspects the EJBs and Java classes, looking for public methods to convert into Web Service operations.
- Creates a `web-services.xml` deployment descriptor file, based on the attributes of the `servicegen` Ant task and introspected information.
- Optionally creates the serialization class that converts the non-built-in data between its XML and Java representations. It also creates XML Schema representations of the Java objects and updates the `web-services.xml` file accordingly. This feature is referred to as *autotyping*.
- Packages all the Web Service components into a Web application WAR file, then packages the WAR and EJB JAR files into a deployable EAR file.

You can also configure default configuration for reliable SOAP messaging, handler chains, and data security (digital signatures and encryption) for a Web Service using `servicegen`.

While you are developing your Web Service, BEA recommends that you create an exploded directory, rather than an EAR file, by specifying a value for the `destEar` attribute of `servicegen` that does *not* have an `.ear` suffix. You can later package the exploded directory into an EAR file when you are ready to deploy the Web Service.

Warning: The serialization class and Java and XML representations generated by the `autotype`, `servicegen`, and `clientgen` Ant tasks cannot be round-tripped. For more information, see "Non-Roundtripping of Generated Data Type Components" on page 6-20.

Note: The fully qualified name of the `servicegen` Ant task is `weblogic.ant.taskdefs.webservices.servicegen.ServiceGenTask`.

Example

```
<servicegen
   destEar="ears/myWebService.ear"
   warName="myWAR.war"
   contextURI="web_services" >
   <service
       ejbJar="jars/myEJB.jar"
       targetNamespace="http://www.bea.com/examples/Trader"
       serviceName="TraderService"
       serviceURI="/TraderService"
       generateTypes="True"
       expandMethods="True" >
   </service>
</servicegen>
```

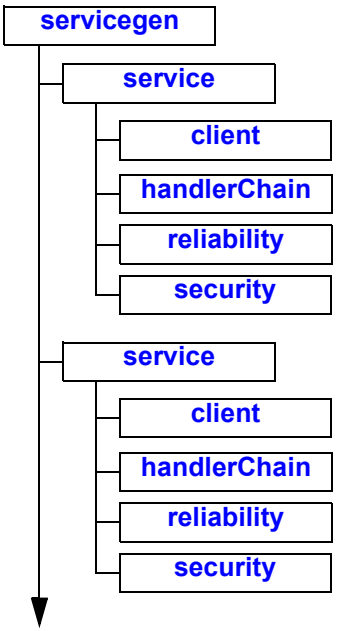
Attributes and Child Elements

The `servicegen` Ant task has four attributes and one child element (`<service>`) for each Web Service you want to define in a single EAR file. You must specify at least one `<service>` element.

The `<service>` element has four optional elements: `<client>`, `<reliability>`, `<handlerChain>`, and `<security>`.

The following graphic describes the hierarchy of the `servicegen` Ant task.

Figure B-1 Element Hierarchy of servicegen Ant Task



servicegen

The `servicegen` Ant task is the main task for automatically generating and assembling all the parts of a Web Service and packaging it into a deployable EAR file.

The following table describes the attributes of the `servicegen` Ant task.

Table B-4 Attributes of the servicegen Ant Task

Attribute	Description	Data Type	Required?
contextURI	Context root of the Web Service. You use this value in the URL that invokes the Web Service. The default value of the `contextURI` attribute is the value of the `warName` attribute.	String	No.
destEar	Pathname of the EAR file or exploded directory which will contain the Web Service and all its components. To create or update an EAR file, use a `.ear` suffix when specifying the EAR file, such as `ears/mywebservice.ear`. If the attribute value does not have a `.ear` suffix, then the `servicegen` task creates an exploded directory. If you specify an EAR file or directory that does not exist, the `servicegen` task creates a new one.	String	Yes
keepGenerated	Specifies whether the `servicegen` Ant task should keep (and thus include in the generated Web Services EAR file) the Java source code of the serialization class for any non-built-in data types used as parameters or return values to the Web Service operations, or whether the `servicegen` Ant task should include only the compiled class file. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.
mergeWithExistingWS	Specifies whether the servicegen Ant task should attempt to merge the generated components into existing Web Services in the EAR file specified by the `destEar` attribute. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.

Table B-4 Attributes of the servicegen Ant Task

Attribute	Description	Data Type	Required?
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No
warName	Name of the WAR file or exploded directory into which the Web Service Web application is written. The WAR file or directory is created at the top level of the EAR file. The default value is a WAR file called `web-services.war`. To specify a WAR file, use a `.war` suffix, such as `mywebserviceWAR.war`. If the attribute value does not have a `.war` suffix, then the `servicegen` task creates an exploded directory.	String	No

service

The `<service>` element describes a single Web Service implemented with either a stateless session EJB or a Java class.

The following table describes the attributes of the `<service>` element of the `servicegen` Ant task. Include one `<service>` element for every Web Service you want to package in a single EAR file.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
ejbJar	JAR file or exploded directory that contains the EJBs that implement the back-end component of a Web Service operation. The `servicegen` Ant task introspects the EJBs to automatically generate all the components.	String	You must specify either the `ejbJar`, `javaClassComponents`, or `JMS*` attribute.
excludeEJBs	Comma-separated list of EJB names for which non-built-in data type components should *not* be generated. If you specify this attribute, the `servicegen` task processes all EJBs *except* those on the list. The EJB names correspond to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor in the EJB JAR file (specified with the `ejbJar` attribute).	String	No. Used only in combination with the `ejbJar` attribute.
expandMethods	Specifies whether the `servicegen` task, when generating the `web-services.xml` file, should create a separate `<operation>` element for each method of the EJB or Java class, or whether the task should implicitly refer to all methods by specifying only one `<operation>` element that contains a `method="*"` attribute. Valid values are `True` and `False`. Default value is `False`.	Boolean	No.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
generateTypes	Specifies whether the `servicegen` task should generate the serialization class and Java representations for non-built-in data types used as parameters or return values. Valid values are `True` and `False`. Default value is `True`. For the list of non-built-in data types for which servicegen can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.	Boolean	No.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request. **Note:** Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it. Valid values are `True` and `False`. Default value is `False`.	Boolean	No.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
includeEJBs	Comma-separated list of EJB names for which non-built-in data type components should be generated. If you specify this attribute, the `servicegen` task processes only those EJBs on the list. The EJB names correspond to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor in the EJB JAR file (specified with the `ejbJar` attribute).	Boolean	No. Used only in combination with the `ejbJar` attribute.
javaClassComponents	Comma-separated list of Java class names that implement the Web Service operation. The Java classes must be compiled and in your CLASSPATH. For example: `javaClassComponents="my.FirstClass,my.SecondClass"` **Note:** Do not include the `.class` extension when specifying the class names. The `servicegen` Ant task introspects the Java classes to automatically generate all the needed components.	String	You must specify either the `ejbJar`, `javaClassComponents`, or `JMS*` attribute.
JMSAction	Specifies whether the client application that invokes this JMS-implemented Web Service sends or receives messages to or from the JMS destination. Valid values are `send` or `receive`. Specify `send` if the client sends messages to the JMS destination and `receive` if the client receives messages from the JMS destination.	String	Yes, if creating a JMS-implemented Web Service.
JMSConnectionFactory	JNDI name of the `ConnectionFactory` used to create a connection to the JMS destination.	String	Yes, if creating a JMS-implemented Web Service.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
JMSDestination	JNDI name of a JMS queue.	String	Yes, if creating a JMS-implemented Web Service.
JMSDestinationType	Type of JMS destination. Currently only one type is supported: Queue. Valid value is `queue`.	String	Yes, if creating a JMS-implemented Web Service.
JMSMessageType	Data type of the single parameter to the send or receive operation. Default value is `java.lang.String`. If you use this attribute to specify a non-built-in data type, and set the `generateTypes` attribute to `True`, be sure the Java representation of this non-built-in data type is in your CLASSPATH.	String	No.
JMSOperationName	Name of the operation in the generated WSDL file. Default value is either `send` or `receive`, depending on the value of the JMSAction attribute.	String	No.
protocol	Protocol over which this Web Service is deployed. Valid values are `http` and `https`. The default value is `http`.	String	No.
serviceName	Name of the Web Service which will be published in the WSDL. **Note:** If you specify more than one `<service>` element in your `build.xml` file that calls `servicegen`, and set the `serviceName` attribute for each element to the same value, `servicegen` attempts to merge the multiple `<service>` elements into a single Web Service.	String	Yes.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
serviceURI	Web Service URI portion of the URL used by client applications to invoke the Web Service. **Note:** Be sure to specify the leading "/", such as `/TraderService`. The full URL to invoke the Web Service will be: *`protocol://host:port/contextURI/serviceURI`* where • *`protocol`* refers to the `protocol` attribute of the `<service>` element • *`host`* refers to the computer on which WebLogic Server is running • *`port`* refers to the port on which WebLogic Server is listening • *`contextURI`* refers to the `contextURI` attribute of the main `servicegen` Ant task • *`serviceURI`* refers to this attribute	String	Yes.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
style	Specifies whether the `servicegen` Ant task should generate RPC-oriented or document-oriented Web Service operations. RPC-oriented WebLogic Web Service operations use SOAP encoding. Document-oriented WebLogic Web Service operations use literal encoding. You can use the following two values to generate document-oriented Web Service operations: `document` and `documentwrapped`. If you specify `document` for this attribute, the resulting Web Service operations take only one parameter. This means that the methods that implement the operations must also have only *one* parameter. In this case, if `servicegen` encounters methods that have more than one parameter, `servicegen` ignores the method and does not generate a corresponding Web Service operation for it. If you specify `documentwrapped`, the resulting Web Service operations can take any number of parameters, although the parameter values will be wrapped into one complex data type in the SOAP messages. If two or more methods of your stateless session EJB or Java class that implement the Web Service have the same number and data type of parameters, and you want the operations to be document-oriented, you must specify `documentwrapped` for this attribute rather than `document`. Valid values for this attribute are `rpc`, `documentwrapped`, and `document`. Default value is `rpc`. **Note:** Because the `style` attribute applies to an entire Web Service, all operations in a single WebLogic Web Service must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web Service.	String	No.

Table B-5 Attributes of the <service> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
typeMappingFile	File that contains additional XML data type mapping information and XML Schema representation of non-built-in data types. The format of the information is the same as the data type mapping information in a `web-services.xml`. Use this attribute if you want to include extra XML data type information in the `<type-mapping>` element of the `web-services.xml` file, in addition to the required XML descriptions of data types used by the EJB or Java class that implements an operation. The `servicegen` task adds the extra information in the specified file to a generated `web-services.xml` file.	String	No.
targetNamespace	The namespace URI of the Web Service.	String	Yes.
useSOAP12	Specifies whether to use SOAP 1.2 as the message format protocol. By default, WebLogic Web Services use SOAP 1.1. If you specify `useSOAP12="True"`, the generated WSDL of the deployed WebLogic Web Service includes *two* ports: the standard port that specifies a binding for SOAP 1.1 as the message format protocol, and a second port that uses SOAP 1.2. Client applications, when invoking the Web Service, can use the second port if they want to use SOAP 1.2 as their message format protocol. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.

client

The optional `<client>` element describes how to create the client JAR file that client applications use to invoke the Web Service. Specify this element only if you want the `servicegen` Ant task to create a client JAR file.

Note: You do not have to create the client JAR file when you assemble your Web Service. You can later use the `clientgen` Ant task to generate the JAR file.

The following table describes the attributes of the `<client>` element.

Table B-6 Attributes of the <client> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
clientJarName	Name of the generated client JAR file. When the `servicegen` task packages the Web Service, it puts the client JAR file in the top-level directory of the Web Service WAR file of the EAR file. Default name is *`serviceName`*`_client.jar`, where *`serviceName`* refers to the name of the Web Service (the `serviceName` attribute) **Note:** If you want a link to the client JAR file to automatically appear in the Web Service Home Page, you should not change its default name.	String	No.
packageName	Package name into which the generated client interfaces and stub files are packaged.	String	Yes.
saveWSDL	When set to `True`, saves the WSDL file of the Web Service in the generated client JAR file. This means that client applications do not need to download the WSDL file every time they create a stub to the Web Service, possibly improving performance of the client because of reduced network usage. Valid values are `True` and `False`. Default value is `True`.	Boolean	No.
useServerTypes	Specifies where the `servicegen` task gets the implementation of any non-built-in Java data types used in a Web Service: either the task generates the Java code or the task gets it from the EAR file that contains the full implementation of the Web Service. Valid values are `True` (use the Java code in the EAR file) and `False`. Default value is `False`. For the list of non-built-in data types for which servicegen can generate data type components, see "Non-Built-In Data Types Supported by servicegen and autotype Ant Tasks" on page 6-16.	Boolean	No.

handlerChain

The optional `<handlerChain>` child element of the `<service>` element adds a handler chain component to the Web Service, and specifies that the handler chain is associated with every operation of the Web Service. A handler chain consists of one or more handlers. For more information on handler chains, see Chapter 12, "Creating SOAP Message Handlers to Intercept the SOAP Message."

Note: Setting this element in `servicegen` associates the handler chain with *every* operation in your Web Service. If you want only some operations to be associated with this handler chain, then you must edit the generated `web-services.xml` file and remove the `handler-chain` attribute of the corresponding `<operation>` element. For details of these elements and attributes, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

The following table describes the attributes of the `<handlerChain>` element.

Table B-7 Attributes of the <handlerChain> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
handlers	Comma separated fully qualified list of Java class names that implement the handlers in the handler chain. You must include at least one class name. **Note:** If the Java class that implements a handler expects initialization parameters, you must edit the generated `web-services.xml` file and add an `<init-params>` child element to the `<handler>` element. For details of these elements, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."	String	Yes.
name	The name of the handler chain. Default value is *`serviceName`*`HandlerChain`, where *`serviceName`* is the value of the `serviceName` attribute of the `<service>` parent element.	String	No.

reliability

The optional `<reliability>` child element of the `<service>` element specifies that every operation of the Web Service can be invoked asynchronously using reliable SOAP messaging. For more information on reliable SOAP messaging, see Chapter 10, "Using Reliable SOAP Messaging."

Note: Setting this element in `servicegen` enables reliable SOAP messaging for *every* operation in your Web Service. If you want only some operations to have reliable SOAP messaging, then you must edit the generated `web-services.xml` file and remove the `<reliable-delivery>` child element of the corresponding `<operation>` element. For details of these elements, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

The following table describes the attributes of the `<reliability>` element.

Table B-8 Attributes of the <reliability> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
duplicateElimination	Specifies whether the WebLogic Web Service operations should ignore duplicate invokes from the same client application. If this attribute is set to `True`, the Web Service persists the message IDs from client applications that invoke the Web Service so that it can eliminate any duplicate invokes. If this values is set to `False`, the Web Service does not keep track of duplicate invokes, which means that if a client retries an invoke, both invokes could return values. Valid values for this attribute are `True` and `False`. The default value is `True`.	Boolean	No.
persistDuration	The default minimum number of seconds that the Web Service should persist the history of a reliable SOAP message (received from the sender that invoked the Web Service) in its storage. The Web Service, after recovering from a WebLogic Server crash, does not dispatch persisted messages that have expired. The default value of this attribute is 60,000 seconds.	Integer	No.

security

The optional `<security>` child element of the `<service>` element adds default data security, such as digital signatures and encryption, to your Web Service. For more information about data security, see Chapter 13, "Configuring Security.".

Note: You can encrypt or digitally sign only the *entire* SOAP message body when you configure data security using the `servicegen` Ant task. If you want to specify particular elements of the SOAP message that are to be digitally signed or encrypted, see "Configuring Message-Level Security: Main Steps" on page 13-9. This section also describes the general security configuration tasks you must perform with the Administration Console before you can successfully invoke your secure Web Service.

The following table describes the attributes of the `<security>` element.

Table B-9 Attributes of the <security> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
enablePasswordAuth	Specifies whether user authentication is enabled or disabled. If enabled, a username token is added to the `<security>` element of the web-services.xml deployment descriptor file. Valid values for the attribute are `True` and `False`. The default value is `False`.	Boolean	No.
encryptKeyName	The name of the key and certificate pair, stored in WebLogic Server's keystore, used to encrypt SOAP message. If you do not specify this attribute, no part of the SOAP message will be encrypted.	String	Only if you want to encrypt the SOAP message.
encryptKeyPass	The password of the key and certificate pair, stored in WebLogic Server's keystore, used to encrypt the SOAP message. If you do not specify this attribute, no part of the SOAP message will be encrypted.	String	Only if you want to encrypt the SOAP message.

Table B-9 Attributes of the <security> Element of the servicegen Ant Task

Attribute	Description	Data Type	Required?
password	Specifies the password used in the username token of the SOAP response message. If you do not specify this attribute, the SOAP response message will not include a username token specification.	String	Only if your SOAP messages require a username token.
signKeyName	The name of the key and certificate pair, stored in WebLogic Server's keystore, used to digitally sign the SOAP message. If you do not specify this attribute, no part of the SOAP message will be digitally signed.	String	Only if you want to digitally sign the SOAP message.
signKeyPass	The password of the key and certificate pair, stored in WebLogic Server's keystore, used to digitally sign the SOAP message. If you do not specify this attribute, no part of the SOAP message will be digitally signed.	String	Only if you want to digitally sign the SOAP message.
username	Specifies the username used in the username token of the SOAP response message. If you do not specify this attribute, the SOAP response message will not include a username token specification.	String	Only if your SOAP messages require a username token.

source2wsdd

The `source2wsdd` Ant task generates a `web-services.xml` deployment descriptor file from the Java source file for a stateless session EJB- or Java class-implemented WebLogic Web Service. You can also specify that the WSDL that describes the Web Service be generated.

The `source2wsdd` Ant task sets the name of the Web Service to the class name of the Java class or EJB that implements the Web Service. This name will also be the public name of the Web Service published in its WSDL.

The `source2wsdd` Ant task does *not* generate data type mapping information for any non-built-in data types used as parameters or return values of the methods of your EJB or Java class If your

EJB or Java class uses non-built-in data types, you must first run the `autotype` Ant task to generate the needed components, then point the `typesInfo` attribute of the `source2wsdd` Ant task to the `types.xml` file generated by the `autotype` Ant task.

If your EJB or Java class refers to other Java class files, be sure to set the `sourcePath` attribute of `source2wsdd` Ant task to the directory that contains them.

Note: The fully qualified name of the `source2wsdd` Ant task is `weblogic.ant.taskdefs.webservices.autotype.JavaSource2DD`.

Example

The following example shows how to generate a `web-services.xml` file, generated into the `ddfiles` directory, for a Java class-implemented Web Service. The information about the non-built-in data types is contained in the `autotype/types.xml` file. The Web Service portion of the URI used to invoke the service is `/MyService`.

```
<source2wsdd
      javaSource="source/MyService.java"
      typesInfo="autotype/types.xml"
      ddFile="ddfiles/web-services.xml"
      serviceURI="/MyService"
/>
```

The following example shows how to generate both a `web-services.xml` file *and* the WSDL file (called `wsdFiles/Temperature.wsdl`) that describes a stateless session EJB-implemented Web Service. Because the `ejbLink` attribute is specified, the `javaSource` attribute must point to the EJB source file. The `source2wsdd` Ant task uses the value of the `ejblink` attribute as the value of the `<ejb-link>` child element of the `<stateless-ejb>` element in the generated `web-services.xml` file.

```
<source2wsdd
       javaSource="source/TemperatureService.java"
       ejbLink="TemperatureService.jar#TemperatureServiceEJB"
       ddFile="ddfiles/web-services.xml"
       typesInfo="autotype/types.xml"
       serviceURI="/TemperatureService"
       wsdlFile="wsdlFiles/Temperature.wsdl"
/>
```

Attributes

The following table describes the attributes of the `source2wsdd` Ant task.

Table B-10 Attributes of the source2wsdd Ant Task

Attribute	Description	Data Type	Required?
ddFile	Full pathname of the Web Services deployment descriptor file (`web-services.xml`) which will contain the generated deployment descriptor information.	String	Yes.
ejblink	Specifies the value of the `<ejb-link>` child element of the `<stateless-ejb>` element in the generated deployment descriptor file for a stateless session EJB-implemented Web Service. Use this attribute only if you are generating the `web-services.xml` file from an EJB. **Note:** The `source2wsdd` Ant task does not use this attribute to determine the EJB which it needs to introspect. Rather, it uses this attribute to determine what value it should use for the `<ejb-link>` element in the generated `web-services.xml` file. Use the `javaSource` attribute to point to the actual EJB source file. The format of this attribute is as follows: *`jar-name#ejb-name`* *`jar-name`* refers to the name of the JAR file, contained within the Web Service EAR file, that contains the stateless session EJB. The name should include pathnames relative to the top level of the EAR file. *`ejb-name`* refers to the name of the stateless session EJB, corresponding to the `<ejb-name>` element in the `ejb-jar.xml` deployment descriptor file in the EJB JAR file. Example: `myapp.jar#StockQuoteBean`	String	No. If you specify this attribute, the required `javaSource` attribute must point to the EJB source file.

Table B-10 Attributes of the source2wsdd Ant Task

Attribute	Description	Data Type	Required?
handlerInfo	Full pathname of the XML file that contains information about the SOAP message handlers and handler chains defined for the Web Service. You must create this file manually. The root element of the file is `<handler-chains>`. For more information about how to populate the file, see "Updating the web-services.xml File with SOAP Message Handler Information" on page 12-19 and Appendix A, "WebLogic Web Service Deployment Descriptor Elements." If you do not specify this attribute, the generated `web-services.xml` file does not contain any SOAP message handlers or handler chain information.	String	No.
ignoreAuthHeader	Specifies that the Web Service ignore the `Authorization` HTTP header in the SOAP request. **Note:** Be careful using this attribute. If you set the value of this attribute to `True`, WebLogic Server *never* authenticates a client application that is attempting to invoke a Web Service, even if access control security constraints have been defined for the EJB, Web Application, or Enterprise Application that make up the Web Service. Or in other words, a client application that does not provide athentication credentials is still allowed to invoke a Web Service that has security constraints defined on it. Valid values are `True` and `False`. Default value is `False`.	Boolean	No.
javaSource	Name of the stateless session EJB or Java source file that implements your Web Service component.	String	Yes.

Table B-10 Attributes of the source2wsdd Ant Task

Attribute	Description	Data Type	Required?
mergeWithExistingWS	Specifies whether the `source2wsdd` Ant task should attempt to merge the generated `web-services.xml` deployment descriptor information with an existing file, specified with the `ddFile` attribute. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
serviceURI	Web Service URI portion of the URL used by client applications to invoke the Web Service. **Note:** Be sure to specify the leading "/", such as `/TraderService`. The value of this attribute becomes the `uri` attribute of the `<web-service>` element in the generated `web-services.xml` deployment descriptor.	String	Yes.
sourcePath	Full pathname of the directory that contains any additional classes referred to by the Java source file specified with the javaSource attribute.	String	No.

Table B-10 Attributes of the source2wsdd Ant Task

Attribute	Description	Data Type	Required?
typesInfo	Name of the file that contains the XML Schema representation and data type mapping information for any non-built-in data types used as parameters or return value of the Web Service. The format of the data type mapping information is the same as that in the `<type-mapping>` element of the `web-services.xml` file. Typically you have already run the `autotype` Ant task to generate this information into a file called `types.xml`.	String	Yes.
wsdlFile	Specifies that, in addition to the `web-services.xml` deployment descriptor, you also want to generate the WSDL that describes the Web Service. Set this value to the name of the output file that will contain the generated WSDL.	String	No.

wsdl2Service

The `wsdl2Service` Ant task takes as input an existing WSDL file and generates:

- the Java interface that represents the implementation of your Web Service based on the WSDL file
- the Java exception class for user-defined exceptions specified in the WSDL file
- an empty Java implementation class
- the `web-services.xml` file that describes the Web Service

The generated Java interface file describes the template for the full Java class-implemented WebLogic Web Service. The template includes full method signatures that correspond to the operations in the WSDL file. You must then write a Java class that implements this interface so that the methods function as you want, following the guidelines in "Implementing a Web Service By Writing a Java Class" on page 5-4. You can generate a skeleton of the implementation class by specifying the `generateImpl="True"` attribute; add the business logic Java code to this class to complete the implementation.

The `wsdl2Service` Ant task generates a Java interface for only one Web Service in a WSDL file (specified by the `<service>` element.) Use the `serviceName` attribute to specify a particular service; if you do not specify this attribute, the `wsdl2Service` Ant task generates a Java interface for the first `<service>` element in the WSDL.

The `wsdl2Service` Ant task does *not* generate data type mapping information for any non-built-in data types used as parameters or return values of the operations in the WSDL file. If the WSDL uses non-built-in data types, you must first run the `autotype` Ant task to generate the data type mapping information, then point the `typeMappingFile` attribute of the `wsdl2Service` Ant task to the `types.xml` file generated by the `autotype` Ant task.

Warning: The `wsdl2Service` Ant task, when generating the `web-services.xml` file for your Web Service, assumes you use the following convention when naming the Java class that implements the generated Java interface:

`packageName.serviceName``Impl`

where *`packageName`* and *`serviceName`* are the values of the similarly-named attributes of the `wsdl2Service` Ant task. The Ant task puts this information in the `class-name` attribute of the `<java-class>` element of the `web-services.xml` file.

If you name your Java implementation class differently, you must manually update the generated `web-services.xml` file accordingly.

Note: The fully qualified name of the `wsdl2Service` Ant task is `weblogic.ant.taskdefs.webservices.wsdl2service.WSDL2Service`.

Example

```
<wsdl2service
      wsdl="wsdls/myService.wsdl"
      destDir="myService/implementation"
      typeMappingFile="autotype/types.xml"
      packageName="example.ws2j.service"
/>
```

Attributes

The following table describes the attributes of the `wsdl2Service` Ant task.

Table B-11 Attributes of the wsdl2Service Ant Task

Attribute	Description	Data Type	Required?
ddFile	Full pathname of the generated Web Services deployment descriptor file (`web-services.xml`) which will contain the deployment descriptor information. If you do not specify this attribute, the `web-services.xml` file is generated in the directory specified by the `destDir` attribute.	String	No.
destDir	The full pathname of the directory that will contain the generated components (`web-services.xml` file and Java interface file that represents the implementation of your Web Service and optional implementation class.)	String	Yes.
generateImpl	Specifies that the `wsdl2Service` Ant task should generate an empty implementation Java class file. The name of the Java class is *`serviceName`*`Impl`, where *`serviceName`* refers to the value of the similarly-named attribute of the `wsdl2Service` Ant task. The name of the generated Java file is *`serviceName`*`Impl.java`, and the file is generated in the directory specified by the `destDir` attribute. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.
keepGenerated	Specifies whether the `wsdl2Service` Ant task should keep (and thus include in directory specified by the `destDir` attribute) the Java source code of the interface that represents your Web service and the user-defined exceptions in the WSDL file. The default behavior is for the Ant task to include only the compiled class files. Valid values for this attribute are `True` and `False`. The default value is `False`.	Boolean	No.

Table B-11 Attributes of the wsdl2Service Ant Task

Attribute	Description	Data Type	Required?
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
packageName	The package name for the generated Java interface file that represents the implementation of your Web Service.	String	Yes.
serviceName	The name of the Web Service in the WSDL file for which a partial WebLogic implementation will be generated. The name of a Web Service in a WSDL file is the value of the `name` attribute of the `<service>` element. If you do not specify this attribute, the `wsdl2Service` Ant task generates a partial implementation for the first `<service>` element it finds in the WSDL file. **Note:** The `wsdl2Service` Ant task generates a partial WebLogic Web Service implementation for only *one* service in a WSDL file. If your WSDL file contains more than one Web Service, then you must run `wsdl2Service` multiple times, changing the value of this attribute each time.	String	No.

Table B-11 Attributes of the wsdl2Service Ant Task

Attribute	Description	Data Type	Required?
typeMappingFile	File that contains data type mapping information for all non-built-in data types referred to by the operations of the Web Service in the WSDL file. The format of the information is the same as the data type mapping information in the `<type-mapping>` element of the `web-services.xml` file. Typically, you first run the `autotype` Ant task (specifying the `wsdl` attribute) against the same WSDL file and generate all the non-built-in data type components. One of the components is a file called `types.xml` that contains the non-built-in data type mapping information and the XML Schema of the non-built-in data types. Set the `typeMappingFile` attribute equal to this file.	String	Required only if the operations of the Web Service in the WSDL file refer to any non-built-in data types.
wsdl	The full path name or URL of the WSDL that describes a Web Service for which a partial WebLogic Web Service implementation will be generated.	String	Yes.

wsdlgen

The `wsdlgen` Ant task generates a WSDL file from the EAR and WAR files that implement your Web Service. The EAR file contains the EJBs that implement your Web Service and the WAR file contains the `web-services.xml` deployment descriptor file.

The fully qualified name of the `wsdlgen` Ant task is `weblogic.ant.taskdefs.webservices.wsdlgen.WSDLGen`.

Example

```
<wsdlgen ear="myapps/myapp.ear"
         warName="myapps/myWAR.war"
         serviceName="myService"
         wsdlFile="wsdls/myService.WSDL"
/>
```

Attributes

The following table describes the attributes of the `wsdlgen` Ant task.

Table B-12 Attributes of the wsdlgen Ant Task

Attribute	Description	Data Type	Required?
ear	Name of an EAR file or exploded directory that contains the WebLogic Web Service implementation for which the WSDL file should be generated.	String	Yes.
defaultEndpoint	Endpoint Web Service URL to be included in the generated WSDL file. The default value is `http://localhost:7001`.	String	No.
overwrite	Specifies whether the components generated by this Ant task should be overwritten if they already exist. If you specify `True`, new components are always generated and any existing components are overwritten. If you specify `False`, the Ant task overwrites only those components that have changed, based on the timestamp of any existing components. Valid values for this attribute is `True` or `False`. The default value is `True`.	Boolean	No.
serviceName	Web Service name for which a corresponding WSDL file should be generated. The Web Service name corresponds to the `<web-service>` element in the `web-services.xml` deployment descriptor file. If you do not specify the `serviceName` attribute, the `wsdlgen` task generates a WSDL file for the first service name found in the `web-services.xml` file.	String	No.
warName	Name of the WAR file that contains the `web-services.xml` deployment descriptor file of your Web Service.	String	Yes.
wsdlFile	Name of the output file that will contain the generated WSDL.	String	Yes.

Equivalent Command-Line Utility

The equivalent command-line utility of the `wsdlgen` Ant task is called `weblogic.webservice.wsdlgen`. The description of the flags of the utility is the same as the description of the Ant task attributes, described in the preceding section.

The `weblogic.webservice.wsdlgen` utility supports the following flags (see the equivalent attribute for a description of the flag):

- `-help (Prints the standard usage message)`
- -version (Prints version information)
- -verbose (Enables verbose output)
- -warName *name*
- -serviceName *name*
- -defaultEndpoint *address*

wspackage

Use the `wspackage` Ant task to:

- package the various components of a WebLogic Web Service into a new deployable EAR file.
- add extra components to an already existing EAR file.

It is assumed that you have already generated the components, which can include:

- The `web-services.xml` deployment descriptor file
- The EJB JAR file that contains the EJBs the implement a Web Service
- The Java class file that implements a Web Service
- A client JAR file that users can download and use to invoke the Web Service
- Implementations of SOAP handlers
- Components for any non-built-in data types used as parameters and return values for the Web Service. These components include the XML and Java representations of the data type and the serialization class that converts the data between its two representations.

Typically you use other Ant tasks, such as `clientgen`, `autotype`, `source2wsdd`, and `wsdl2Service`, to generate the preceding components.

When you use the `wspackage` Ant task to add additional components to an existing EAR file, be sure you specify the `overwrite="false"` attribute to ensure that none of the existing components are overwritten. Use the `output` attribute to specify the full pathname of the existing EAR file.

Note: The fully qualified name of the `wspackage` Ant task is `weblogic.ant.taskdefs.webservices.wspackage.WSPackage`.

Example

The following example shows how to use the `wspackage` Ant task to package WebLogic Web Service components into a new EAR file:

```
<wspackage
      output="ears/myWebService.ear"
      contextURI="web_services"
      codecDir="autotype"
      webAppClasses="example.ws2j.service.SimpleTest"
      ddFile="ddfiles/web-services.xml"
/>
```

The following example shows how to add additional components to an existing EAR file called `ears/myWebService.ear`.

```
<wspackage
      output="ears/myWebService.ear"
      overwrite="false"
      filesToEar="myEJB2.jar"
/>
```

Attributes

The following table describes the attributes of the `wspackage` Ant task.

Table B-13 Attributes of the wspackage Ant Task

Attribute	Description	Data Type	Required?
codecDir	Name of the directory that contains the serialization classes for any non-built-in data types used as parameters or return values in your Web Service.	String	No.
contextURI	Context root of the Web Service. You use this value in the URL that invokes the Web Service. The default value of the `contextURI` attribute is the value of the `warName` attribute.	String	No.
ddFile	Full pathname of an existing Web Services deployment descriptor file (`web-services.xml`). If you do not specify this attribute, it is assumed that you are using the `wspackage` Ant task to add additional components to an existing EAR file. In this case, you must also specify `overwrite="false"` and point the `output` attribute to the existing EAR file.	String	No.
filesToEar	Comma-separated list of files to be packaged in the root directory of the EAR. Use this attribute to specify the EJB JAR files that implement a Web Service, as well as any other supporting EJB JAR files.	String	No.
filesToWar	Comma-separated list of additional files, such as the client JAR file, to be packaged in the root directory of the Web Service's Web application.	String	No.

Table B-13 Attributes of the wspackage Ant Task

Attribute	Description	Data Type	Required?
output	Pathname of the EAR file or exploded directory which will contain the Web Service and all its components. If you are using the `wspackage` Ant task to add additional components to an existing EAR file, this attribute specifies the full pathname of the existing file. To create or update an EAR file, use a `.ear` suffix when specifying the EAR file, such as `ears/mywebservice.ear`. If the attribute value does not have a `.ear` suffix, then the `wspackage` task creates an exploded directory. If you specify an EAR file or directory that does not exist, the `wspackage` task creates a new one.	String	Yes
overwrite	Specifies whether you want the components of an existing EAR file or directory to be overwritten. The components include the `web-services.xml` file, serialization class, client JAR files, and so on. Valid values for this attribute are `True` and `False`. The default value is `True`. If you specify `False`, the `wspackage` Ant task attempts to merge the contents of the EAR file/directory and information in the `web-services.xml` file. If you are using the wspackage Ant task to add additional components to an existing EAR file, you must specify `overwrite="false"`.	Boolean	No
utilJars	Comma-separated list of files that should be packaged in the `WEB-INF/lib` directory of the Web Service Web application.	String	No.

Table B-13 Attributes of the wspackage Ant Task

Attribute	Description	Data Type	Required?
warName	Name of the WAR file into which the Web Service is written. The WAR file is created at the top level of the EAR file. The default value is `web-services.war`.	String	No
webAppClasses	Comma-separated list of class files that should be packaged in the `WEB-INF/classes` directory of the Web Service's Web application. Use this attribute to specify the Java class that implements a Web Service, SOAP handler classes, and so on.	String	No.

APPENDIX C

source2wsdd Tag Reference

The following topics describe the `source2wsdd` Javadoc tags:

- "Overview of Using source2wsdd Tags" on page C-1
- "@wlws:webservice" on page C-2
- "@wlws:operation" on page C-5
- "@wlws:part partname" on page C-7
- "@wlws:exclude" on page C-11

Overview of Using source2wsdd Tags

The `source2wsdd` Ant task generates a `web-services.xml` deployment descriptor file from the Java source file for a stateless session EJB- or Java class-implemented WebLogic Web Service. The `web-services.xml` deployment descriptor file contains information that describes one or more WebLogic Web Services. This information includes details about the back-end components that implement the operations of a Web Service, the non-built-in data types used as parameters and return values, the SOAP message handlers that intercept SOAP messages, and so on. As is true for all deployment descriptors, `web-services.xml` is an XML file.

This chapter describes the optional Javadoc tags you can include in the Java source file that the `source2wsdd` Ant task uses to automatically generate the `web-services.xml` file. If your Java source file does not contain any Javadoc tags, the `source2wsdd` Ant task makes a best guess when adding elements to the file, such as the name of the Web Service, the operations that should be exposed, and so on. If, however, you want more control over the generated

web-services.xml file, use the source2wsdd Javadoc tags to specify exactly what your Web Service looks like.

There are three source2wsdd Javadoc tags:

- @wlws:webservice, used in the Javadoc for the class that implements your Web Service.
- @wlws:operation, used in the Javadoc for a method that you want to expose as a Web Service operation.
- @wlws:part partname used in the Javadoc for a method that has been exposed as an operation and you want to customize the description of its parameters and return values.

Each tag has a set of attributes which correspond to the appropriate element in the web-services.xml file that it is describing.

@wlws:webservice

The source2wsdd Ant task uses the @wlws:webservice tag to populate the <web-service> element of the generated web-services.xml file.

You specify the @wlws:webservice tag in the Javadoc of the class that implements your Web Service.

The following example shows how to use the @wlws:webservice tag in the Javadoc that documents a Java class:

```
/**
 * PurchaseOrderService - a service to show different features of
 * Weblogic Web Services.
 *
 * @wlws:webservice
 *    targetNamespace="http://www.bea.com/po-service/"
 *    name="PurchaseOrderService"
 *    portName="POPort"
 *    portTypeName="POPort"
 *    protocol="https"
*/
public class POService {
  ...
}
```

In the example, the `POService` Java class is the back-end component that implements a Web Service whose name is `PurchaseOrderService` (specified with the `name` attribute of the `@wls:webservice` tag). The `<port>` and `<portType>` elements in the generated WSDL for the Web Service are both `POPort`. Finally, client applications access the Web Service using HTTPS rather than the default HTTP.

The following table lists all the attributes of the `@wlws:webservice` tag.

Table C-1 Attributes of the @wlws:webservice source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
Name	The name of the Web Service, published in the WSDL.	The name of the class in the annotated source code.
portName	Name of the `<port>` child element of the `<service>` element of the dynamically generated WSDL of this Web Service.	The name of the Web Service with `Port` appended. For example, if the name of this Web Service is `TraderService`, the default port name is `TraderServicePort`.
portTypeName	Name of the default `<portType>` element in the dynamically generated WSDL of this Web service.	The name of this Web Service with `Port` appended. For example, if the name of this Web service is `TraderService`, the default portType name is `TraderServicePort`.
protocol	Protocol over which the Web Service is invoked. Valid values are `http` and `https`.	Default value is `http`.

Table C-1 Attributes of the @wlws:webservice source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
Style	Specifies whether the Web Service has RPC-oriented or document-oriented Web Service operations. RPC-oriented WebLogic Web Service operations use SOAP encoding. Document-oriented WebLogic Web Service operations use literal encoding. You can use the following two values to generate document-oriented Web Service operations: `document` and `documentwrapped`. If you specify `document` for this attribute, the resulting Web Service operations take only one parameter. This means that the methods that implement the operations must also have only *one* parameter. In this case, if `source2wsdd` encounters methods that have more than one parameter, `source2wsdd` ignores the method and does not generate a corresponding Web Service operation for it. If you specify `documentwrapped`, the resulting Web Service operations can take any number of parameters, although the parameter values will be wrapped into one complex data type in the SOAP messages. If two or more methods of your stateless session EJB or Java class that implement the Web Service have the same number and data type of parameters, and you want the operations to be document-oriented, you must specify `documentwrapped` for this attribute rather than `document`. Valid values for this attribute are `rpc`, `documentwrapped`, and `document`.Because the `style` attribute applies to an entire Web Service, all operations in a single WebLogic Web Service must be either RPC-oriented or documented-oriented; WebLogic Server does not support mixing the two styles within the same Web Service.	Default value is `rpc`.

Table C-1 Attributes of the @wlws:webservice source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
targetNameSpace	The namespace URI of the Web Service.	Default value is `http://tempuri.org`.
Uri	Web Service URI portion of the URL used by client applications to invoke the Web Service. **Note:** Be sure to specify the leading "/", such as `/TraderService`. **Note:** You can also specify the URI using the `serviceURI` attribute of the `source2wsdd` Ant task. If you specify the URI in both places, and they are different from each other, the URI specified by the `@wlws:webservice` Javadoc tag takes precedence.	Although you are not required to specify the `Uri` attribute of the `@wlws:webservice` tag, you *are* required to specify the `serviceURI` of the `source2wsdd` Ant task. This means that there is no default value and that at some point you must specify the Web Service URI.

@wlws:operation

The `source2wsdd` Ant task uses the `@wlws:operation` tag to populate the corresponding `<operation>` element in the generated `web-services.xml` file.

By default, every public method of the Java class or EJB that implements a Web Service is exposed as an operation in the generated WSDL. The `source2wsdd` Ant task uses information from the method, such as its signature, to populate the `<operation>` element; use the `@wlws:operation` Javadoc tag to change some of the default information, such as the operation name. Use the @wlws:exclude tag to specify that a public method not be exposed as a Web Service operation.

You specify the `@wlws:operation` tag in the Javadoc of the method that implements the operation.

The following example shows how to use the `@wlws:operation` tag in the Javadoc that documents a method:

```
/**
   * A one way call. Client will not wait for this method to
   * complete.
   *
   * Note: A one way call must have a void return type.
```

```
 *
 * @wlws:operation
 *       invocation-style="one-way"
 *       Name="sendTime"
 */
public void oneWayCall( long time ){
  .....
}
```

In the example, the `oneWayCall` method implements an operation of a Web Service called `sendTime`, which is the published name of the operation in the WSDL. The operation is one-way, which means that the client application does not receive a return value.

The following table lists all the attributes of the `@wlws:operation` tag.

Table C-2 Attributes of the @wlws:operation source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
Handler-chain	Name of the SOAP message handler chain that, together with the method, implements the operation. The name of this attribute corresponds to the `name` attribute of the appropriate `<handler-chain>` element in the file that contains the SOAP handler chain information. You must write this file manually and specify its location with the `handlerInfo` attribute of the `source2wsdd` Ant task.	If you do not specify this attribute, no handler-chain information is added to the `<operation>` element in the `web-services.xml` file.

Table C-2 Attributes of the @wlws:operation source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
invocation-style	Specifies whether the operation both receives a SOAP request and sends a SOAP response (request-response), or whether the operation only receives a SOAP request but does *not* send back a SOAP response (one-way). Valid values are `request-response` and `one-way`. **Note:** If the back-end component that implements this operation is a method of a stateless session EJB or Java class and you set this attribute to `one-way`, the method must return `void`.	Default value is `request-response`.
Name	The name of the operation. This is the name that is published in the WSDL of the Web Service.	The name of the method in the Java source file.

@wlws:part partname

The `source2wsdd` Ant task uses the `@wlws:part` tag to populate the corresponding `<param>` and `<return-param>` elements that describe the parameters and return values for the operation in the generated `web-services.xml` file.

Every public method of the Java class or EJB that implements a Web Service is exposed as an operation in the generated WSDL. The `source2wsdd` Ant task uses information from the method signature to determine basic information about the parameters and return value of the operation. If, however, you want to change some of this default information, specify the `@wlws:part` tag in the Javadoc of the method. In particular, use the attributes of the tag when you want:

- The name of the parameters and return values in the generated WSDL to be different from those of the method that implements the operation.
- To map a parameter to a name in the SOAP header request or response.
- To use out or in-out parameters.
- To explicitly specify the XML and Java represenation of the data type of the paramter or return value.

Use the `@wlws:part` tag in the Javadoc of the method that implements the operation. Specify the name of the parameter right after the tag and before the attributes, as shown:

```
@wlws:part paramName attribute="value"
```

To specify the return value, use the hard-coded word *return*, as shown:

```
@wlws:part return attribute="value"
```

The following example shows how to use the `@wlws:part` tag in the Javadoc that documents a method:

```
/**
   * operation with headers
   *
   * @wlws:part addressInHeader location="header"
   * @wlws:part dataInHeader location="header"
   *
   * @wlws:part return location="body"
   */
  public BaseData methodWithHeaders( String addressInHeader,
      int idInBody, BaseData dataInHeader ){

    dataInHeader.setAddress( addressInHeader );
    dataInHeader.setId( idInBody );

    return dataInHeader;
  }
```

In the example, when a client application invokes the `methodWithHeaders` operation, the `addressInHeader` and `dataInHeader` input parameters are located in the header of the SOAP request. When WebLogic Server responds to the invocation of the operation, the return value is located in the body of the SOAP response.

The following table lists all the attributes of the `@wlws:part` tag.

Table C-3 Attributes of the @wlws:part source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
class-name	Java class name of the Java representation of the data type of the inpur or return parameter.	The data type of the parameter or return value of the operation. **Note:** If the mapping between the XML and Java representations of the parameter or return value is ambiguous (such as `xsd:int` mapping to either the `int` Java primitive or `java.lang.Integer`), and you do not specify this attribute, WebLogic Server makes its best as to which mapping is correct.

Table C-3 Attributes of the @wlws:part source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
location	Part of the request or response SOAP message (header, body, or attachment) that contains the value of the input or return parameter. Valid values for this attribute are `Body`, `Header`, or `attachment`. If you specify `Body`, the value of the input or return parameter is contained in the SOAP Body (of either the request or response, depending on whether the parameter is input or return). If you specify `Header`, the value contained in a SOAP Header element whose name is the value of the `type` attribute. If you specify `attachment`, the value of the parameter is contained in the SOAP Attachment rather than the SOAP envelope. As specified by the JAX-RPC specification, only the following Java data types can be contained in the SOAP Attachment: • `java.awt.Image` • `java.lang.String` • `javax.mail.internet.MimeMultiport` • `javax.xml.transform.Source` • `javax.activation.DataHandler`	The default value is `Body`.
name	The name of the parameter. This is the name that is published in the WSDL of the Web Service in the `<part>` element.	For input parameters, the default value is the name of the parameter in the method's signature. The default value of the return parameter is `results`.

Table C-3 Attributes of the @wlws:part source2wsdd Tag

Attribute Name	Description	Default Value if Attribute is Not Specified
style	Style of the input parameter: either a standard input parameter, an out parameter used as a return value, or an in-out parameter for both inputting and outputting values. Valid values for this attribute are `in`, `out`, and `inout`. If you specify a parameter as `out` or `inout`, the Java class of the parameter in the back-end component's method must implement the `javax.xml.rpc.holders.Holder` interface.	The default value is `in`.
type	XML Schema data type of the parameter. If you specify this attribute of the `@wlws:part` tag, you must also specify a `types.xml` file using the `typesInfo` attribute of the `source2wsdd` Ant task. You must also ensure that the XML Schema data type you specify for this tag exists in the `types.xml` file, and that the two element names match exactly. If the `source2wsdd` Ant task does not find the name of this XML Schema data type in the `types.xml` file, the Ant task generates its own data type mapping information, which could lead to incorrect behavior of your Web Service.	If you do not specify this attribute, the XML data type is based on the Java data type of the parameter.

@wlws:exclude

The `source2wsdd` Ant task uses the `@wlws:exclude` tag to exclude public methods of the Java source file from the list of generated Web Service operations.

By default, every public method of the Java class or EJB that implements a Web Service is exposed as an operation in the generated WSDL. If you do not want to expose a public method, you must explicitly add the `@wlws:exclude` tag to the method's Javadoc.

The following example shows how to use the `@wlws:exclude` tag:

```
/**
 * A public method that is not exposed as a Web Service operation.
 *
 * @wlws:exclude
 */
public void dontExposeThisMethod(){

}
```

In the example, the `source2wsdd` will not add the public method `dontExposeThisMethod()` to the list of Web Service operations in the generated `webservices.xml` file, and thus it will also not appear in the generated WSDL file.

The `@wlws:exclude` tag does not have any attributes.

APPENDIX D

Customizing WebLogic Web Services

The following sections describe how to customize your WebLogic Web Service by updating the Web application deployment descriptor files of your Web Service WAR file:

- "Publishing a Static WSDL File" on page D-1
- "Creating a Custom WebLogic Web Service Home Page" on page D-2

Publishing a Static WSDL File

By default, WebLogic Server dynamically generates the WSDL of a WebLogic Web Service, based on the contents of its `web-services.xml` deployment descriptor file. See "WebLogic Web Services Home Page and WSDL URLs" on page 6-21 for details on getting the URL of the dynamically generated WSDL.

You can, however, include a static version of the WSDL file in the Web Services EAR file and publish its URL as the public description of your Web Service. One reason for publishing a static WSDL is to be able to add more custom documentation than what the dynamically generated WSDL contains.

Warning: If you publish a static WSDL as the public description of your Web Service, you must always ensure that it remains up to date with the actual Web Service. In other words, if you change your Web Service, you must also manually change the static WSDL to reflect the changes you made to your Web Service. One advantage of using the dynamic WebLogic-generated WSDL is that it is always up to date.

To include a static WSDL file in your Web Services EAR file and publish it, rather than the dynamically generated WSDL, to the Web, follow these steps:

1. Un-JAR the WebLogic Web Services EAR file and then the WAR file that contains the `web-services.xml` file.
2. Put the static WSDL file in a directory of the exploded Web application. This procedure assumes you put the file at the top-level directory.
3. Update the `web.xml` file of the Web application, adding a `<mime-mapping>` element to map the extension of your WSDL file to an XML mime type.

   For example, if the name of your static WSDL file is `myService.wsdl`, the corresponding entry in the `web.xml` file is as follows:

   ```
   <mime-mapping>
           <extension>wsdl</extension>
           <mime-type>text/xml</mime-type>
   </mime-mapping>
   ```

4. Re-JAR the Web Services WAR and EAR files.
5. Invoke the static WSDL file using the standard URL to invoke a static file in a Web application.

   For example, use the following URL to invoke the `myService.wsdl` file in a Web application that has a context root of `web_services`:

   `http://`*`host`*`:`*`port`*`/web_services/myService.wsdl`

Creating a Custom WebLogic Web Service Home Page

Every WebLogic Web Service has a default Home Page that contains links to view the WSDL of the Web Service, test the service, download the client JAR file, and view the SOAP requests and responses of a client application invoking the Web Service. See "WebLogic Web Services Home Page and WSDL URLs" on page 6-21 for details.

WebLogic Server dynamically generates the Web Services Home page and thus it cannot be customized. If you want to create your own custom Home Page, add an HTML or JSP file to the Web Services WAR file. For more information on creating JSPs, see *Programming WebLogic JSP* at http://e-docs.bea.com/wls/docs81/jsp/index.html.

APPENDIX E

Assembling a WebLogic Web Service Manually

The following sections provide information about assembling a WebLogic Web Service manually:

- "Overview of Assembling a WebLogic Web Service Manually" on page E-1
- "Assembling a WebLogic Web Service Manually: Main Steps" on page E-2
- "Understanding the web-services.xml File" on page E-2
- "Creating the web-services.xml File Manually: Main Steps" on page E-3
- "Examining Different Types of web-services.xml Files" on page E-9

Overview of Assembling a WebLogic Web Service Manually

Assembling a WebLogic Web Service refers to gathering all the components of the service (such as the EJB JAR file, the SOAP message handler classes, and so on), generating the `web-services.xml` deployment descriptor file, and packaging everything into an Enterprise Application EAR file that can be deployed on WebLogic Server.

Typically you never assemble a WebLogic Web Service manually, because the procedure is complex and time-consuming. Rather, use the WebLogic Ant tasks such as `servicegen`, `autotype`, `source2wsdd`, and so on to automatically generate all the needed components and package them into a deployable EAR file.

If, however, your Web Service is so complex that the Ant tasks are not able to generate the needed components, or you want full control over all aspects of the Web Service assembly, then use this chapter as a guide to assembling the Web Service manually.

Assembling a WebLogic Web Service Manually: Main Steps

1. Package or compile the back-end components that implement the Web Service into their respective packages. For example, package stateless session EJBs into an EJB JAR file and Java classes into class files.

   For detailed instructions, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/programming/environment.html.

2. Create the Web Service deployment descriptor file (`web-services.xml`).

   For a description of the `web-services.xml` file, see "Understanding the web-services.xml File" on page E-2. For detailed steps for creating the file manually, see "Creating the web-services.xml File Manually: Main Steps" on page E-3.

3. If your Web Service uses non-built-in data types, create all the needed components, such as the serialization class.

   For detailed information on creating these components manually, see Chapter 11, "Using Non-Built-In Data Types."

4. Package all components into a deployable EAR file.

   When packaging the EAR file manually, be sure to put the correct Web Service components into a Web application WAR file. For details about the WAR and EAR file hierarchy, see "The Web Service EAR File Package" on page 6-16. For instructions, see *Developing WebLogic Server Applications* at http://e-docs.bea.com/wls/docs81/programming/environment.html.

Understanding the web-services.xml File

The `web-services.xml` deployment descriptor file contains information that describes one or more WebLogic Web Services, such as the back-end components that implement the Web Service; the non-built-in data types used as parameters and return values; the SOAP message handlers that intercept SOAP messages; and so on. As is true for all deployment descriptors, `web-services.xml` is an XML file.

Based on the contents of the `web-services.xml` deployment descriptor file, WebLogic Server dynamically generates the WSDL of a deployed WebLogic Web Service. See "WebLogic Web Services Home Page and WSDL URLs" on page 6-21 for details on getting the URL of the dynamically generated WSDL.

A single WebLogic Web Service consists of one or more operations; you can implement each operation using methods of different back-end components and SOAP message handlers. For example, an operation might be implemented with a single method of a stateless session EJB or with a combination of SOAP message handlers and a method of a stateless session EJB.

A single `web-services.xml` file contains a description of at least one, and maybe more, WebLogic Web Services.

If you are assembling a Web Service manually (necessary, for example, is the service uses SOAP message handlers and handler chains), you need to create the `web-services.xml` file manually. If you assemble a WebLogic Web Service with the `servicegen` Ant task, you do not need to create the `web-services.xml` file manually, because the Ant task generates one for you based on its introspection of the EJBs, the attributes of the Ant task, and so on.

Even if you need to manually assemble a Web Service, you can use the `servicegen` Ant task to create a basic template, and then use this document to help you update the generated `web-services.xml` with the extra information that `servicegen` does not provide.

Creating the web-services.xml File Manually: Main Steps

The `web-services.xml` deployment descriptor file describes one or more WebLogic Web Service. The file includes information about the operations that make up the Web Services, the back-end components that implement the operations, data type mapping information about non-built-in data types used as parameters and return values of the operations, and so on. See "Examining Different Types of web-services.xml Files" on page E-9 for complete examples of `web-services.xml` files that describe different kinds of WebLogic Web Services. You can use any text editor to create the `web-services.xml` file.

For detailed descriptions of each element described in this section, see Appendix A, "WebLogic Web Service Deployment Descriptor Elements."

The following example shows a simple `web-services.xml` file; the procedure following the example describes the main steps to create the file.

```
<web-services>
     <web-service name="stockquotes" targetNamespace="http://example.com"
                  uri="/myStockQuoteService">
          <components>
               <stateless-ejb name="simpleStockQuoteBean">
                    <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
               </stateless-ejb>
          </components>
          <operations>
               <operation method="getLastTradePrice"
```

```
                        component="simpleStockQuoteBean" />
        </operations>
    </web-service>
</web-services>
```

To create the preceding `web-services.xml` file manually:

1. Create the root `<web-services>` element which contains all other elements:

   ```
   <web-services>
   ...
   </web-services>
   ```

2. If one or more of your Web Services include SOAP message handlers to intercept SOAP messages, create a `<handler-chains>` child element of the `<web-services>` root element and include all the relevant child elements to describe the handlers in the handler chain, the order in which they should be invoked, and so on. For details, see "Updating the web-services.xml File with SOAP Message Handler Information" on page 12-19.

3. For each Web Service you want to define, follow these steps:

   a. Create a `<web-service>` child element of the `<web-services>` element. Use the `name`, `targetNamespace`, and `uri` attributes to specify the name of the Web Service, its target namespace, and the URI that clients will use to invoke the Web Service, as shown in the following example:

      ```
      <web-service name="stockquote"
                   targetNamespace="http://example.com"
                   uri="myStockQuoteService">
      ...
      </web-service>
      ```

      To specify that the operations in your Web Service are all document-oriented, use the `style="document"` attribute. The default value of the `style` attribute is `rpc`, which means the operations are all RPC-oriented.

   b. Create a `<components>` child element of the `<web-service>` element that lists the back-end components that implement the operations of the Web Service. For details, see "Creating the <components> Element" on page E-5.

   c. If the operations in your Web Service use non-built-in data types as parameters or return values, add data type mapping information by creating `<types>` and `<type-mapping>` child elements of the `<web-service>` element. For details, see "Creating the Data Type Mapping File" on page 11-10.

Note: You do not have to perform this step if the operations of your Web Service use only built-in data types as parameters or return values. See "Supported Built-In Data Types" on page 5-15 for a list of the supported built-in data types.

d. Create an `<operations>` child element of the `<web-service>` element that lists the operations that make up the Web Service:

```
<operations xmlns:xsd="http://www.w3.org/2001/XMLSchema">
....
</operations>
```

e. Within the `<operations>` element, list the operations defined for the Web Service. For details, see "Creating <operation> Elements" on page E-6.

Creating the <components> Element

Use the `<components>` child element of the `<web-service>` element to list and describe the back-end components that implement the operations of a Web Service. Each back-end component has a `name` attribute that you later use when describing the operation that the component implements.

Note: If you are creating a SOAP message handler-only type of Web Service in which handlers and handler chains do all the work and never execute a back-end component, you do not specify a `<components>` element in the `web-services.xml` file. For all other types of Web Services you *must* declare a `<components>` element.

You can list one of the following types of back-end components:

- `<stateless-ejb>`

  This element describes a stateless EJB back-end component. Use either the `<ejb-link>` child element to specify the name of the EJB and the JAR file where it is located or the `<jndi-name>` child element to specify the JNDI name of the EJB, as shown in the following example:

```
<components>
  <stateless-ejb name="simpleStockQuoteBean">
     <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
  </stateless-ejb>
</components>
```

- `<java-class>`

  This element describes a Java class back-end component. Use the `class-name` attribute to specify the fully qualified path name of the Java class, as shown in the following example:

```
        <components>
           <java-class name="customClass"
                       class-name="myclasses.MyOwnClass" />
        </components>
```

Creating <operation> Elements

The <operation> element describes how the public operations of a WebLogic Web Service are implemented. (The public operations are those that are listed in the Web Service's WSDL and are executed by a client application that invokes the Web Service.) The following example shows an <operation> declaration:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
   <params>
      <param name="in1" style="in" type="xsd:string" location="Header"/>
      <param name="in2" style="in" type="xsd:int"  location="Header"/>
      <return-param name="result" type="xsd:string"  location="Header"/>
   </params>
</operation>
```

Typically, every instance of an <operation> element in the web-services.xml file includes the name attribute which translates into the public name of the Web Service operation. The only exception is when you use the method="*" attribute to specify all methods of an EJB or Java class in a single <operation> element; in this case, the public name of the operation is the name of the method.

Use the attributes of the <operation> element in combination to specify different kinds of operations. For details, see "Specifying the Type of Operation" on page E-6.

Use the <params> element to optionally group together the parameters and return value of the operation. For details, see "Specifying the Parameters and Return Value of the Operation" on page E-8.

Specifying the Type of Operation

Use the attributes of the <operation> element in different combination to identify the type of operation, the type of component that implements it, whether it is a one-way operation, and so on.

Note: For clarity, the examples in this section do not declare any parameters.

The following examples show how to declare a variety of different operations:

- To specify that an operation is implemented with just a method of a stateless session EJB, use the `name`, `component`, and `method` attributes, as shown in the following example:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote">
</operation>
```

- To specify with a single `<operation>` element that you want to include all the methods of an EJB or Java class, use the `method="*"` attribute; in this case, the public name of the operation is the name of the method:

```
<operation component="simpleStockQuoteBean"
           method="*">
</operation>
```

- To specify that an operation only receives data and does not return anything to the client application, add the `invocation-style` attribute:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote(java.lang.String)"
           invocation-style="one-way">
</operation>
```

  The example also shows how to specify the full signature of a method with the `method` attribute. You only need to specify the full signature of a method if your EJB or Java class overloads the method and you thus need to unambiguously declare which method you are exposing as a Web Service operation.

- To specify that an operation is implemented with a SOAP message handler chain and a method of a stateless session EJB, use the `name`, `component`, `method`, and `handler-chain` attributes:

```
<operation name="getQuote"
           component="simpleStockQuoteBean"
           method="getQuote"
           handler-chain="myHandler">
</operation>
```

- To specify that an operation is implemented with just a SOAP message handler chain, use just the `name` and `handler-chain` attributes:

```
<operation name="justHandler"
           handler-chain="myHandler">

</operation>
```

Specifying the Parameters and Return Value of the Operation

Use the <params> element to explicitly declare the parameters and return values of the operation.

You do not have to explicitly list the parameters or return values of an operation. If an <operation> element does not have a <params> child element, WebLogic Server introspects the back-end component that implements the operation to determine its parameters and return values. When generating the WSDL of the Web Service, WebLogic Server uses the names of the corresponding method's parameters and return value.

You explicitly list an operation's parameters and return values when you need to:

- Make the name of the parameters and return values in the generated WSDL different from those of the method that implements the operation.
- Map a parameter to a name in the SOAP header request or response.
- Use out or in-out parameters.

Use the <param> child element of the <params> element to specify a single input parameter and the <return-param> child element to specify the return value. You must list the input parameters in the same order in which they are defined in the method that implements the operation. The number of <param> elements must match the number of parameters of the method. You can specify only one <return-param> element.

Use the attributes of the <param> and <return-param> elements to specify the part of the SOAP message where parameter is located (the body or header), the type of the parameter (in, out, or in-out), and so on. You must always specify the XML Schema data type of the parameter using the type attribute. The following examples show a variety of input and return parameters.

- To specify that a parameter is a standard input parameter, located in the header of the request SOAP message, use the style and location attributes as shown:

  ```
  <param name="inparam" style="in"
         location = "Header" type="xsd:string" />
  ```

- Out and in-out parameters enable an operation to return more than one return value (in addition to using the standard <return-value> element.) The following sample <param> element shows how to specify that a parameter is an in-out parameter, which means that it acts as both an input and output parameter:

  ```
  <param name="inoutparam" style="inout"
         type="xsd:int" />
  ```

  Because the default value of the location attribute is Body, both the input and output parameter values are found in the body of the SOAP message.

- The following example shows how to specify a standard return value located in the header of the response SOAP message:

```
<return-param name="result" location="Header"
              type="xsd:string" />
```

Optionally use the `<fault>` child element of the `<params>` element to specify your own Java exception that is thrown if there is an error while invoking the operation. This exception will be thrown in addition to the `java.rmi.RemoteException` exception. For example:

```
<fault name="MyServiceException"
       class-name="my.exceptions.MyServiceException" />
```

Examining Different Types of web-services.xml Files

The following sections provide examples of `web-services.xml` files for various types of WebLogic Web Services:

- EJB Component Web Service with Built-In Data Types
- EJB Component Web Service with Non-Built-In Data Types
- EJB Component and SOAP Message Handler Chain Web Service
- SOAP Message Handler Chain Web Service

EJB Component Web Service with Built-In Data Types

One kind of WebLogic Web Service is implemented using a stateless session EJB whose parameters and return values are one of the built-in data types. The following Java interface is an example of such an EJB:

```
public interface SimpleStockQuoteService extends javax.ejb.EJBObject {
 public float getLastTradePrice(String ticker) throws java.rmi.RemoteException;
}
```

The following example shows a possible `web-services.xml` deployment descriptor for a Web Service implemented with this sample EJB:

```
<web-services>
     <web-service name="stockquotes" targetNamespace="http://example.com"
                  uri="/myStockQuoteService">
          <components>
               <stateless-ejb name="simpleStockQuoteBean">
                    <ejb-link path="stockquoteapp.jar#StockQuoteBean" />
               </stateless-ejb>
```

```
        </components>
        <operations>
             <operation method="getLastTradePrice"
                        component="simpleStockQuoteBean" />
        </operations>
    </web-service>
</web-services>
```

The example shows a Web Service called `stockquotes`. The Web Service is implemented with a stateless session EJB whose `<ejb-name>` in the `ejb-jar.xml` file is `StockQuoteBean` and is packaged in the EJB JAR file called `stockquoteapp.jar`. The internal name of this component is `simpleStockQuoteBean`. The Web Service has one operation, called `getLastTradePrice`, the same as the EJB method name. The input and output parameters are inferred from the method signature and thus do not need to be explicitly specified in the `web-services.xml` file.

Note: The `servicegen` Ant task does not include the methods of `EJBObject` when generating the list of operations in the `web-services.xml` file.

The previous example shows how to explicitly list an operation of a Web Service. You can, however, implicitly expose all the public methods of an EJB by including just one `<operation method="*">` element, as shown in the following example:

```
<operations>
     <operation method="*"
                component="simpleStockQuoteBean" />
</operations>
```

If your Web Service supports only HTTPS, then use the `protocol` attribute of the `<web-service>` element, as shown in the following example:

```
<web-service name="stockquotes"
             targetNamespace="http://example.com"
             uri="/myStockQuoteService"
             protocol="https" >
...
</web-service>
```

EJB Component Web Service with Non-Built-In Data Types

A more complex type of Web Service is one whose operations take non-built-in data types as parameters or return values. Because these non-built-in data types do not directly map to a XML/SOAP data type, you must describe the data type in the `web-services.xml` file.

For example, the following interface describes an EJB whose two methods return a `TradeResult` object:

```
public interface Trader extends EJBObject {
   public TradeResult buy (String stockSymbol, int shares)
       throws RemoteException;
   public TradeResult sell (String stockSymbol, int shares)
       throws RemoteException;
}
```

The `TradeResult` class looks like the following:

```
public class TradeResult implements Serializable {

  private int    numberTraded;
  private String stockSymbol;

  public TradeResult() {}

  public TradeResult(int nt, String ss) {
    numberTraded = nt;
    stockSymbol  = ss;
  }

  public int getNumberTraded() { return numberTraded; }
  public void setNumberTraded(int numberTraded) {
     this.numberTraded = numberTraded; }

  public String getStockSymbol() { return stockSymbol; }
  public void setStockSymbol(String stockSymbol) {
     this.stockSymbol = stockSymbol; }
}
```

The following `web-services.xml` file describes a Web Service implemented with this EJB:

```
<web-services>

  <web-service name="TraderService"
               uri="/TraderService"
               targetNamespace="http://www.bea.com/examples/Trader">

    <types>
      <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                  xmlns:stns="java:examples.webservices"
                  attributeFormDefault="qualified"
                  elementFormDefault="qualified"
```

```
                targetNamespace="java:examples.webservices">
        <xsd:complexType name="TradeResult">
           <xsd:sequence><xsd:element maxOccurs="1" name="stockSymbol"
                                      type="xsd:string" minOccurs="1">
                         </xsd:element>
                         <xsd:element maxOccurs="1" name="numberTraded"
                                       type="xsd:int" minOccurs="1">
                          </xsd:element>
           </xsd:sequence>
        </xsd:complexType>
      </xsd:schema>
     </types>

    <type-mapping>
      <type-mapping-entry
            deserializer="examples.webservices.TradeResultCodec"
            serializer="examples.webservices.TradeResultCodec"
            class-name="examples.webservices.TradeResult"
            xmlns:p1="java:examples.webservices"
            type="p1:TradeResult" >
     </type-mapping-entry>
    </type-mapping>

    <components>
      <stateless-ejb name="ejbcomp">
        <ejb-link path="trader.jar#TraderService" />
      </stateless-ejb>
    </components>

    <operations>
      <operation method="*" component="ejbcomp">
      </operation>
    </operations>

  </web-service>

</web-services>
```

In the example, the `<types>` element uses XML Schema notation to describe the XML representation of the `TradeResult` data type. The `<type-mapping>` element contains an entry for each data type described in the `<types>` element (in this case there is just one: `TradeResult`.) The `<type-mapping-entry>` lists the serialization class that converts the data between XML and Java, as well as the Java class file used to create the Java object.

EJB Component and SOAP Message Handler Chain Web Service

Another type of Web Service is implemented with both a stateless session EJB back-end component and a SOAP message handler chain that intercepts the request and response SOAP message. The following sample `web-services.xml` file describes such a Web Service:

```
<web-services>
    <handler-chains>
        <handler-chain name="submitOrderCrypto">
            <handler class-name="com.example.security.EncryptDecrypt">
                <init-params>
                   <init-param name="elementToDecrypt" value="credit-info" />
                   <init-param name="elementToEncrypt" value="order-number" />
                </init-params>
            </handler>
        </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com" name="myorderproc"
                 uri="myOrderProcessingService">
        <components>
            <stateless-ejb name="orderbean">
               <ejb-link path="myEJB.jar#OrderBean" />
            </stateless-ejb>
        </components>
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
            <operation name="submitOrder" method="submit"
                       component="orderbean"
                       handler-chain="submitOrderCrypto" >
               <params>
                 <param name="purchase-order" style="in" type="xsd:anyType" />
                 <return-param name="order-number" type="xsd:string" />
               </params>
            </operation>
        </operations>
    </web-service>
</web-services>
```

The example shows a Web Service that includes a SOAP message handler-chain called `submitOrderCrypto` used for decrypting and encrypting information in the SOAP request and response messages. The handler chain includes one handler, implemented with the `com.example.security.EncryptDecrypt` Java class. The handler takes two initialization parameters that specify the elements in the SOAP message that need to be decrypted and encrypted.

The Web Service defines one stateless session EJB back-end component called `orderbean`.

The `submitOrder` operation shows how to combine a handler-chain with a back-endback-end component by specifying the `method`, `component`, and `handler-chain` attributes in combination. When a client application invokes the `submitOrder` operation, the `submitOrderCrypto` handler chain first processes the SOAP request, decrypting the credit card information. The handler chain then invokes the `submit()` method of the `orderbean` EJB, passing it the modified parameters from the SOAP message, including the `purchase-order` input parameter. The `submit()` method then returns an `order-number`, which is encrypted by the handler chain, and the handler chain finally sends a SOAP response with the encrypted information to the client application that originally invoked the `submitOrder` operation.

SOAP Message Handler Chain Web Service

You can also implement a WebLogic Web Service with just a SOAP message handler chain and never invoke a back-end component. This type of Web Service might be useful, for example, as a front end to an existing workflow processing system. The handler chain simply takes the SOAP message request and hands it over to the workflow system, which performs all the further processing.

The following sample web-services.xml file describes such a Web Service:

```
<web-services>
    <handler-chains>
       <handler-chain name="enterWorkflowChain">
           <handler class-name="com.example.WorkFlowEntry">
               <init-params>
                   <init-param name="workflow-eng-jndi-name"
                               value="workflow.entry" />
                </init-params>
           </handler>
       </handler-chain>
    </handler-chains>

    <web-service targetNamespace="http://example.com"
                 name="myworkflow" uri="myWorkflowService">
        <operations xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
           <operation name="enterWorkflow"
                      handler-chain="enterWorkflowChain"
                      invocation-style="one-way" />
        </operations>
    </web-service>
</web-services>
```

The example shows a Web Service that includes one SOAP message handler chain, called `enterWorkflowChain`. This handler chain has one handler, implemented with the Java class `com.example.WorkFlowEntry`, that takes as an initialization parameter the JNDI name of the existing workflow system.

The Web Service defines one operation called `enterWorkflow`. When a client application invokes this operation, the `enterWorkflowChain` handler chain takes the SOAP message request and passes it to the workflow system running on WebLogic Server whose JNDI name is `workflow.entry`. The operation is defined as asynchronous one-way, which means that the client application does not receive a SOAP response.

Note that because the `enterWorkflow` operation does *not* specify the `method` and `component` attributes, no back-end component is ever invoked directly by the Web Service. This also means that the `web-services.xml` file does not need to specify a `<components>` element.